$\langle Version \rangle \equiv$
```
2.2.1
```
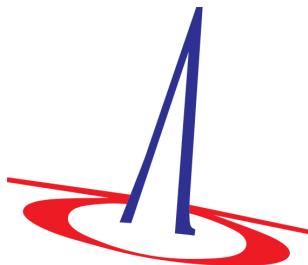$\langle Date \rangle \equiv$
```
June 3 2014
```

WHIZARD[1]

Wolfgang Kilian,[2] Thorsten Ohl,[3] Jürgen Reuter[4]

Version 2.2.1, June 3 2014
with contributions from: Fabian Bach, Felix Braam, David Gordo
Gomez, Sebastian Schmidt, Marco Sekulla, Christian Speckner,
Daniel Wiesler

---

[1]The original meaning of the acronym is *W, Higgs, Z, And Respective Decays*. The current program is much more than that, however.

[2]e-mail: `kilian@physik.uni-siegen.de`

[3]e-mail: `ohl@physik.uni-wuerzburg.de`

[4]e-mail: `juergen.reuter@desy.de`

Abstract

WHIZARD is an application of the VAMP algorithm: Adaptive multi-channel integration and event generation. The bare VAMP library is augmented by modules for Lorentz algebra, particles, phase space, etc., such that physical processes with arbitrary complex final states [well, in principle. . .] can be integrated and *unweighted* events be generated.

Contents

13

14

Chapter 1

Changes

For a comprehensive list of changes confer the ChangeLog file or the subversion log.

Chapter 2

Tools

This part contains modules needed for auxiliary purposes:

limits Compile-time (integer) constants: fixed array sizes, input field lengths, and such. Any module that uses such constants has to access them via **limits**.

mpi90 Parallel execution (currently disabled!). This module contains dummy replacements for the message passing interface subroutines.

diagnostics Error and diagnostic message handling. Any messages and errors issued by WHIZARD functions are handled by the subroutines in this module, if possible.

files Files and auxiliary routines that do not belong anywhere else.

permutations Handle permutations of integers.

unix_args Parse the command-line arguments and options. This part is system-dependent, and we provide a replacement if the functions are not available.

2.1 Preliminaries

The WHIZARD file header:

⟨*File header*⟩≡
```
  ! WHIZARD ⟨Version⟩ ⟨Date⟩
  !
  ! Copyright (C) 1999-2014 by
  !     Wolfgang Kilian <kilian@physik.uni-siegen.de>
  !     Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
  !     Juergen Reuter <juergen.reuter@desy.de>
  !
  !     with contributions from
  !     Christian Speckner <cnspeckn@googlemail.com>
  !     and  Fabian Bach, Felix Braam, Sebastian Schmidt, Daniel Wiesler
  !
  ! WHIZARD is free software; you can redistribute it and/or modify it
  ! under the terms of the GNU General Public License as published by
```

```
! the Free Software Foundation; either version 2, or (at your option)
! any later version.
!
! WHIZARD is distributed in the hope that it will be useful, but
! WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
! GNU General Public License for more details.
!
! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This file has been stripped of most comments.  For documentation, refer
! to the source 'whizard.nw'
```

We are strict with our names:

⟨*Standard module head*⟩≡
```
    implicit none
    private
```

This is the way to envoke the kinds module (not contained in this source)

⟨*Use kinds*⟩≡
```
    use kinds, only: default !NODEP!
```

And we make heavy use of variable-length strings

⟨*Use strings*⟩≡
```
    use iso_varying_string, string_t => varying_string !NODEP!
```

Some parameters (buffer sizes etc.) are hardcoded. They are collected in this module:

⟨`limits.f90`⟩≡
  ⟨*File header*⟩

  module limits

    use iso_fortran_env, only: iostat_end, iostat_eor !NODEP!
  ⟨*Use kinds*⟩

  ⟨*Standard module head*⟩

  ⟨*Limits: public parameters*⟩

  end module limits

The version string is used for checking files. Note that the string length MUST NOT be changed, because reading binary files relies on it.

⟨*Limits: public parameters*⟩≡
```
    integer, parameter, public :: VERSION_STRLEN = 255
    character(len=VERSION_STRLEN), parameter, public :: &
        & VERSION_STRING = "WHIZARD version ⟨Version⟩ (⟨Date⟩)"
```

We collect format string for the different numerical output formats here.

⟨*Limits: public parameters*⟩+≡
```
    character(*), parameter, public :: FMT_19 = "ES19.12"
    character(*), parameter, public :: FMT_18 = "ES18.11"
```

```
character(*), parameter, public :: FMT_17 = "ES17.10"
character(*), parameter, public :: FMT_16 = "ES16.9"
character(*), parameter, public :: FMT_15 = "ES15.8"
character(*), parameter, public :: FMT_14 = "ES14.7"
character(*), parameter, public :: FMT_13 = "ES13.6"
character(*), parameter, public :: FMT_12 = "ES12.5"
```

2.2 Parallelization

This section has been removed, it was never used even in WHIZARD1. Parallelization should be reimplemented from scratch.

2.3 File utilities

This module provides miscellaneous tools associated with strings and files:

- Finding a free unit

- Selecting an output unit (stdout if undefined)

- Upper and lower case for strings

- Formatting a number for TEX output.

⟨`file_utils.f90`⟩≡
  ⟨*File header*⟩

```
  module file_utils

    use iso_fortran_env, only: stdout => output_unit !NODEP!
```
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use limits, only: MIN_UNIT, MAX_UNIT !NODEP!
    use, intrinsic :: iso_c_binding !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*File utils: public*⟩

  ⟨*File utils: interfaces*⟩

```
  contains
```

  ⟨*File utils: procedures*⟩

```
  end module file_utils
```

2.3.1 Finding an I/O unit

Fortran 95 (even Fortran 2003) has no notion of implicit I/O units. Therefore, we have to find a free unit by trial and error.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MIN_UNIT = 11, MAX_UNIT = 99
```

⟨*File utils: public*⟩≡
```
public :: free_unit
```

⟨*File utils: procedures*⟩≡
```
function free_unit () result (unit)
  integer :: unit
  logical :: exists, is_open
  integer :: i, status
  do i = MIN_UNIT, MAX_UNIT
     inquire (unit=i, exist=exists, opened=is_open, iostat=status)
     if (status == 0) then
        if (exists .and. .not. is_open) then
           unit = i; return
        end if
     end if
  end do
  unit = -1
end function free_unit
```

⟨*Use file utils*⟩≡
```
use file_utils !NODEP!
```

Return the given unit, if present, otherwise the default STDOUT unit.

⟨*File utils: public*⟩+≡
```
public :: output_unit
```

⟨*File utils: procedures*⟩+≡
```
function output_unit (unit) result (u)
  integer, intent(in), optional :: unit
  integer :: u
  if (present (unit)) then
     u = unit
  else
     u = stdout
  end if
end function output_unit
```

2.3.2   Deleting a file

Fortran does not contain a command for deleting a file. Here, we provide a subroutine that deletes a file if it exists. We do not handle the subtleties, so we assume that it is writable if it exists.

⟨*File utils: public*⟩+≡
```
public :: delete_file
```

⟨*File utils: procedures*⟩+≡
```
subroutine delete_file (name)
  character(*), intent(in) :: name
  logical :: exist
  integer :: u
  inquire (file = name, exist = exist)
  if (exist) then
     u = free_unit ()
     open (unit = u, file = name)
```

```
      close (u, status = "delete")
    end if
  end subroutine delete_file
```

2.3.3 String auxiliary functions

These are, unfortunately, not part of Fortran.

⟨*File utils: public*⟩+≡
```
    public :: upper_case
    public :: lower_case
```

⟨*File utils: interfaces*⟩≡
```
    interface upper_case
       module procedure upper_case_char, upper_case_string
    end interface
    interface lower_case
       module procedure lower_case_char, lower_case_string
    end interface
```

⟨*File utils: procedures*⟩+≡
```
    function upper_case_char (string) result (new_string)
      character(*), intent(in) :: string
      character(len(string)) :: new_string
      integer :: pos, code
      integer, parameter :: offset = ichar('A')-ichar('a')
      do pos = 1, len (string)
         code = ichar (string(pos:pos))
         select case (code)
         case (ichar('a'):ichar('z'))
            new_string(pos:pos) = char (code + offset)
         case default
            new_string(pos:pos) = string(pos:pos)
         end select
      end do
    end function upper_case_char

    function lower_case_char (string) result (new_string)
      character(*), intent(in) :: string
      character(len(string)) :: new_string
      integer :: pos, code
      integer, parameter :: offset = ichar('a')-ichar('A')
      do pos = 1, len (string)
         code = ichar (string(pos:pos))
         select case (code)
         case (ichar('A'):ichar('Z'))
            new_string(pos:pos) = char (code + offset)
         case default
            new_string(pos:pos) = string(pos:pos)
         end select
      end do
    end function lower_case_char

    function upper_case_string (string) result (new_string)
```

```
    type(string_t), intent(in) :: string
    type(string_t) :: new_string
    new_string = upper_case_char (char (string))
  end function upper_case_string

  function lower_case_string (string) result (new_string)
    type(string_t), intent(in) :: string
    type(string_t) :: new_string
    new_string = lower_case_char (char (string))
  end function lower_case_string
```

Quote underscore characters for use in TEX output.

⟨*File utils: public*⟩+≡
```
    public :: quote_underscore
```

⟨*File utils: procedures*⟩+≡
```
    function quote_underscore (string) result (quoted)
      type(string_t) :: quoted
      type(string_t), intent(in) :: string
      type(string_t) :: part
      type(string_t) :: buffer
      buffer = string
      quoted = ""
      do
        call split (part, buffer, "_")
        quoted = quoted // part
        if (buffer == "")  exit
        quoted = quoted // "\_"
      end do
    end function quote_underscore
```

Convert a FORTRAN string into a zero terminated C string.

⟨*File utils: public*⟩+≡
```
    public :: string_f2c
```

⟨*File utils: interfaces*⟩+≡
```
    interface string_f2c
       module procedure string_f2c_char, string_f2c_var_str
    end interface string_f2c
```

⟨*File utils: procedures*⟩+≡
```
    pure function string_f2c_char (i) result (o)
    character(*), intent(in) :: i
    character(kind=c_char, len=len (i) + 1) :: o
      o = i // c_null_char
    end function string_f2c_char

    pure function string_f2c_var_str (i) result (o)
    type(string_t), intent(in) :: i
    character(kind=c_char, len=len (i) + 1) :: o
      o = char (i) // c_null_char
    end function string_f2c_var_str
```

2.3.4 Formatting numbers

Format a number with n significant digits for use in TeX documents.

⟨*File utils: public*⟩+≡

```
public :: tex_format
```

⟨*File utils: procedures*⟩+≡

```
function tex_format (rval, n_digits) result (string)
  type(string_t) :: string
  real(default), intent(in) :: rval
  integer, intent(in) :: n_digits
  integer :: e, n, w, d
  real(default) :: absval
  real(default) :: mantissa
  character :: sign
  character(20) :: format
  character(80) :: cstr
  n = min (abs (n_digits), 16)
  if (rval == 0) then
     string = "0"
  else
     absval = abs (rval)
     e = log10 (absval)
     if (rval < 0) then
        sign = "-"
     else
        sign = ""
     end if
     select case (e)
     case (:-3)
        d = max (n - 1, 0)
        w = max (d + 2, 2)
        write (format, "('(F',I0,'.',I0,',A,I0,A)')")  w, d
        mantissa = absval * 10._default ** (1 - e)
        write (cstr, fmt=format)  mantissa, "\times 10^{", e - 1, "}"
     case (-2:0)
        d = max (n - e, 1 - e)
        w = max (d + e + 2, d + 2)
        write (format, "('(F',I0,'.',I0,')')")  w, d
        write (cstr, fmt=format)  absval
     case (1:2)
        d = max (n - e - 1, -e, 0)
        w = max (d + e + 2, d + 2, e + 2)
        write (format, "('(F',I0,'.',I0,')')")  w, d
        write (cstr, fmt=format)  absval
     case default
        d = max (n - 1, 0)
        w = max (d + 2, 2)
        write (format, "('(F',I0,'.',I0,',A,I0,A)')")  w, d
        mantissa = absval * 10._default ** (- e)
        write (cstr, fmt=format)  mantissa, "\times 10^{", e, "}"
     end select
     string = sign // trim (cstr)
  end if
end function tex_format
```

Write a number for use in Metapost code:

⟨*File utils: public*⟩+≡
```
public :: mp_format
```

⟨*File utils: procedures*⟩+≡
```
function mp_format (rval) result (string)
  type(string_t) :: string
  real(default), intent(in) :: rval
  character(16) :: tmp
  write (tmp, "(G16.8)")  rval
  string = lower_case (trim (adjustl (trim (tmp))))
end function mp_format
```

2.3.5 Auxiliary stuff

Write a separator line.

⟨*File utils: public*⟩+≡
```
public :: write_separator
public :: write_separator_double
```

⟨*File utils: procedures*⟩+≡
```
subroutine write_separator (u)
  integer, intent(in) :: u
  write (u, "(A)")  repeat ("-", 72)
end subroutine write_separator

subroutine write_separator_double (u)
  integer, intent(in) :: u
  write (u, "(A)")  repeat ("=", 72)
end subroutine write_separator_double
```

⟨*File utils: public*⟩+≡
```
public :: write_indent
```

⟨*File utils: procedures*⟩+≡
```
subroutine write_indent (unit, indent)
  integer, intent(in) :: unit
  integer, intent(in), optional :: indent
  if (present (indent)) then
     write (unit, "(1x,A)", advance="no")  repeat ("  ", indent)
  end if
end subroutine write_indent
```

Conditional format string.

⟨*File utils: public*⟩+≡
```
public :: pac_fmt
```

⟨*File utils: procedures*⟩+≡
```
    subroutine pac_fmt (fmt, fmt_orig, fmt_pac, pacify)
      character(*), intent(in) :: fmt_orig, fmt_pac
      character(*), intent(out) :: fmt
      logical, intent(in), optional :: pacify
      logical :: pacified
      pacified = .false.
      if (present (pacify)) pacified = pacify
      if (pacified) then
         fmt = fmt_pac
      else
         fmt = fmt_orig
      end if
    end subroutine pac_fmt
```

2.4 Unit tests

Here, we provide functionality for automated unit tests. Each test is required to produce output which is compared against a reference file. If the two are identical, we signal success. Otherwise, we signal failure and write the output to a file.

⟨`unit_tests.f90`⟩≡
  ⟨*File header*⟩

```
  module unit_tests
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*Tests: public*⟩

  ⟨*Tests: parameters*⟩

  ⟨*Tests: types*⟩

  ⟨*Tests: interfaces*⟩

```
  contains
```

  ⟨*Tests: procedures*⟩

```
  end module unit_tests
```

2.4.1 Parameters

Building blocks of file names:

⟨*Tests: parameters*⟩≡
```
    character(*), parameter :: ref_prefix = "ref-output/"
```

```
character(*), parameter :: ref = ".ref"

character(*), parameter :: err_prefix = "err-output/"
character(*), parameter :: err = ".out"
```

2.4.2 Type for storing test results

We store the results of the individual unit tests in a linked list. Here is the entry:

⟨*Tests: public*⟩≡
```
public :: test_results_t
```

⟨*Tests: types*⟩≡
```
type :: test_result_t
   logical :: success = .false.
   type(string_t) :: name
   type(string_t) :: description
   type(test_result_t), pointer :: next => null ()
end type test_result_t

type :: test_results_t
   private
   type(test_result_t), pointer :: first => null ()
   type(test_result_t), pointer :: last => null ()
   integer :: n_success = 0
   integer :: n_failure = 0
 contains
 ⟨Tests: test results: TBP⟩
end type test_results_t
```

Add a test result.

⟨*Tests: test results: TBP*⟩≡
```
procedure, private :: add => test_results_add
```

⟨*Tests: procedures*⟩≡
```
subroutine test_results_add (list, name, description, success)
  class(test_results_t), intent(inout) :: list
  character(len=*), intent(in) :: name
  character(len=*), intent(in) :: description
  logical, intent(in) :: success
  type(test_result_t), pointer :: result
  allocate (result)
  result%success = success
  result%name = name
  result%description = description
  if (associated (list%first)) then
     list%last%next => result
  else
     list%first => result
  end if
  list%last => result
  if (success) then
     list%n_success = list%n_success + 1
```

```
      else
          list%n_failure = list%n_failure + 1
      end if
   end subroutine test_results_add
```

Display the current state.

⟨*Tests: test results: TBP*⟩+≡
```
    procedure, private :: write => test_results_write
```

⟨*Tests: procedures*⟩+≡
```
    subroutine test_results_write (list, u)
      class(test_results_t), intent(in) :: list
      integer, intent(in) :: u
      type(test_result_t), pointer :: result
      write (u, "(A)")  "*** Test Summary ***"
      if (list%n_success > 0) then
          write (u, "(2x,A)")  "Success:"
          result => list%first
          do while (associated (result))
             if (result%success)  write (u, "(4x,A,': ',A)")  &
                   char (result%name), char (result%description)
             result => result%next
          end do
      end if
      if (list%n_failure > 0) then
          write (u, "(2x,A)")  "Failure:"
          result => list%first
          do while (associated (result))
             if (.not. result%success)  write (u, "(4x,A,': ',A)")  &
                   char (result%name), char (result%description)
             result => result%next
          end do
      end if
      write (u, "(A,I0)")  "Total   = ", list%n_success + list%n_failure
      write (u, "(A,I0)")  "Success = ", list%n_success
      write (u, "(A,I0)")  "Failure = ", list%n_failure
      write (u, "(A)")  "*** End of test Summary ***"
    end subroutine test_results_write
```

Return true if all tests were successful (or no test).

⟨*Tests: test results: TBP*⟩+≡
```
    procedure, private :: report => test_results_report
```

⟨*Tests: procedures*⟩+≡
```
    subroutine test_results_report (list, success)
      class(test_results_t), intent(in) :: list
      logical, intent(out) :: success
      success = list%n_failure == 0
    end subroutine test_results_report
```

Delete the list.

⟨*Tests: test results: TBP*⟩+≡
```
    procedure, private :: final => test_results_final
```

⟨*Tests: procedures*⟩+≡

```fortran
    subroutine test_results_final (list)
      class(test_results_t), intent(inout) :: list
      type(test_result_t), pointer :: result
      do while (associated (list%first))
         result => list%first
         list%first => result%next
         deallocate (result)
      end do
      list%last => null ()
      list%n_success = 0
      list%n_failure = 0
    end subroutine test_results_final
```

2.4.3 Wrapup

This will write results, report status, and finalize. This is the only method which we need to access from outside.

⟨*Tests: test results: TBP*⟩+≡

```fortran
    procedure :: wrapup => test_results_wrapup
```

⟨*Tests: procedures*⟩+≡

```fortran
    subroutine test_results_wrapup (list, u, success)
      class(test_results_t), intent(inout) :: list
      integer, intent(in) :: u
      logical, intent(out), optional :: success
      call list%write (u)
      if (present (success))  call list%report (success)
      call list%final ()
    end subroutine test_results_wrapup
```

2.4.4 Tool for Unit Tests

This procedure takes a test routine as an argument. It runs the test, output directed to a temporary file. Then, it compares the file against a reference file.

The test routine must take the output unit as argument.

⟨*Tests: interfaces*⟩≡

```fortran
    abstract interface
       subroutine unit_test (u)
         integer, intent(in) :: u
       end subroutine unit_test
    end interface
```

The test routine can print to screen and, optionally, to a logging unit.

⟨*Tests: public*⟩+≡

```fortran
    public :: test
```

⟨*Tests: procedures*⟩+≡

```
subroutine test (test_proc, name, description, u_log, results)
  procedure(unit_test) :: test_proc
  character(*), intent(in) :: name
  character(*), intent(in) :: description
  integer, intent(in) :: u_log
  type(test_results_t), intent(inout) :: results
  integer :: u_test, u_ref, u_err
  logical :: exist
  character(256) :: buffer1, buffer2
  integer :: iostat1, iostat2
  logical :: success
  write (*, "(A)", advance="no")  "Running test: " // name
  write (u_log, "(A)")  "Test: " // name
  u_test = free_unit ()
  open (u_test, status="scratch", action="readwrite")
  call test_proc (u_test)
  rewind (u_test)
  inquire (file=ref_prefix//name//ref, exist=exist)
  if (exist) then
     u_ref = free_unit ()
     open (u_ref, file=ref_prefix//name//ref, status="old", action="read")
     COMPARE_FILES: do
        read (u_test, "(A)", iostat=iostat1)  buffer1
        read (u_ref, "(A)", iostat=iostat2)  buffer2
        if (iostat1 /= iostat2) then
           success = .false.
           exit COMPARE_FILES
        else if (iostat1 < 0) then
           success = .true.
           exit COMPARE_FILES
        else if (buffer1 /= buffer2) then
           success = .false.
           exit COMPARE_FILES
        end if
     end do COMPARE_FILES
     close (u_ref)
  else
     write (*, "(A)", advance="no") " ... no reference output available"
     write (u_log, "(A)") "  No reference output available."
     success = .false.
  end if
  if (success) then
     write (*, "(A)") " ... success."
     write (u_log, "(A)")  " Success."
  else
     write (*, "(A)") " ... failure.  See: " // err_prefix//name//err
     write (u_log, "(A)")  " Failure."
     rewind (u_test)
     u_err = free_unit ()
     open (u_err, file=err_prefix//name//err, &
           action="write", status="replace")
     WRITE_OUTPUT: do
        read (u_test, "(A)", end=1) buffer1
```

```
          write (u_err, "(A)")  trim (buffer1)
        end do WRITE_OUTPUT
1       close (u_err)
      end if
      close (u_test)
      call results%add (name, description, success)
    end subroutine test
```

2.5   Operating-system interface

For specific purposes, we need direct access to the OS (system calls). This is,
of course, system dependent. The current version is valid for GNU/Linux; we
expect to use a preprocessor for this module if different OSs are to be supported.

   The current implementation lacks error handling.

⟨os_interface.f90⟩≡
  ⟨File header⟩

  module os_interface

    use iso_c_binding !NODEP!
  ⟨Use strings⟩
  ⟨Use file utils⟩
    use system_dependencies !NODEP!
    use limits, only: DLERROR_LEN, ENVVAR_LEN !NODEP!
    use diagnostics !NODEP!
    use unit_tests

  ⟨Standard module head⟩

  ⟨OS interface: public⟩

  ⟨OS interface: types⟩

  ⟨OS interface: interfaces⟩

  contains

  ⟨OS interface: procedures⟩

  ⟨OS interface: tests⟩

  end module os_interface

2.5.1   Path variables

This is a transparent container for storing user-defined path variables.

⟨OS interface: public⟩≡
    public :: paths_t

⟨*OS interface: types*⟩≡
```
type :: paths_t
   type(string_t) :: prefix
   type(string_t) :: exec_prefix
   type(string_t) :: bindir
   type(string_t) :: libdir
   type(string_t) :: includedir
   type(string_t) :: datarootdir
   type(string_t) :: localprefix
   type(string_t) :: libtool
   type(string_t) :: lhapdfdir
end type paths_t
```

⟨*OS interface: public*⟩+≡
```
public :: paths_init
```

⟨*OS interface: procedures*⟩≡
```
subroutine paths_init (paths)
  type(paths_t), intent(out) :: paths
  paths%prefix = ""
  paths%exec_prefix = ""
  paths%bindir = ""
  paths%libdir = ""
  paths%includedir = ""
  paths%datarootdir = ""
  paths%localprefix = ""
  paths%libtool = ""
  paths%lhapdfdir = ""
end subroutine paths_init
```

2.5.2 System dependencies

We store all potentially system- and user/run-dependent data in a transparent container. This includes compiler/linker names and flags, file extensions, etc.

⟨*OS interface: public*⟩+≡
```
public :: os_data_t
```

⟨*OS interface: types*⟩+≡
```
type :: os_data_t
   logical :: use_libtool
   logical :: use_testfiles
   type(string_t) :: fc
   type(string_t) :: fcflags
   type(string_t) :: fcflags_pic
   type(string_t) :: fc_src_ext
   type(string_t) :: cc
   type(string_t) :: cflags
   type(string_t) :: cflags_pic
   type(string_t) :: obj_ext
   type(string_t) :: ld
   type(string_t) :: ldflags
   type(string_t) :: ldflags_so
   type(string_t) :: ldflags_static
```

```
        type(string_t) :: ldflags_hepmc
        type(string_t) :: ldflags_hoppet
        type(string_t) :: shlib_ext
        type(string_t) :: makeflags
        type(string_t) :: prefix
        type(string_t) :: exec_prefix
        type(string_t) :: bindir
        type(string_t) :: libdir
        type(string_t) :: includedir
        type(string_t) :: datarootdir
        type(string_t) :: whizard_omega_binpath
        type(string_t) :: whizard_includes
        type(string_t) :: whizard_ldflags
        type(string_t) :: whizard_libtool
        type(string_t) :: whizard_modelpath
        type(string_t) :: whizard_models_libpath
        type(string_t) :: whizard_susypath
        type(string_t) :: whizard_gmlpath
        type(string_t) :: whizard_cutspath
        type(string_t) :: whizard_texpath
        type(string_t) :: whizard_testdatapath
        type(string_t) :: whizard_modelpath_local
        type(string_t) :: whizard_models_libpath_local
        type(string_t) :: whizard_omega_binpath_local
        type(string_t) :: whizard_circe2path
        type(string_t) :: whizard_beamsimpath
        type(string_t) :: whizard_mulipath
        type(string_t) :: pdf_builtin_datapath
        logical :: event_analysis = .false.
        logical :: event_analysis_ps  = .false.
        logical :: event_analysis_pdf = .false.
        type(string_t) :: latex
        type(string_t) :: mpost
        type(string_t) :: gml
        type(string_t) :: dvips
        type(string_t) :: ps2pdf
      end type os_data_t
```

Since all are allocatable strings, explicit initialization is necessary.

⟨*Limits: public parameters*⟩+≡
```
      integer, parameter, public :: ENVVAR_LEN = 1000
```

⟨*OS interface: public*⟩+≡
```
      public :: os_data_init
```

⟨*OS interface: procedures*⟩+≡
```
      subroutine os_data_init (os_data, paths)
        type(os_data_t), intent(out) :: os_data
        type(paths_t), intent(in), optional :: paths
        character(len=ENVVAR_LEN) :: home
        type(string_t) :: localprefix, local_includes
        os_data%use_libtool = .true.
        inquire (file = "TESTFLAG", exist = os_data%use_testfiles)
        call get_environment_variable ("HOME", home)
        if(present(paths)) then
```

```
         if (paths%localprefix == "") then
            localprefix = trim (home) // "/.whizard"
         else
            localprefix = paths%localprefix
         end if
      else
         localprefix = trim (home) // "/.whizard"
      end if
      local_includes = localprefix // "/lib/whizard/mod/models"
      os_data%whizard_modelpath_local = localprefix // "/share/whizard/models"
      os_data%whizard_models_libpath_local = localprefix // "/lib/whizard/models"
      os_data%whizard_omega_binpath_local = localprefix // "/bin"
      os_data%fc           = DEFAULT_FC
      os_data%fcflags      = DEFAULT_FCFLAGS
      os_data%fcflags_pic  = DEFAULT_FCFLAGS_PIC
      os_data%fc_src_ext   = DEFAULT_FC_SRC_EXT
      os_data%cc           = DEFAULT_CC
      os_data%cflags       = DEFAULT_CFLAGS
      os_data%cflags_pic   = DEFAULT_CFLAGS_PIC
      os_data%obj_ext      = DEFAULT_OBJ_EXT
      os_data%ld           = DEFAULT_LD
      os_data%ldflags      = DEFAULT_LDFLAGS
      os_data%ldflags_so   = DEFAULT_LDFLAGS_SO
      os_data%ldflags_static = DEFAULT_LDFLAGS_STATIC
      os_data%ldflags_hepmc  = DEFAULT_LDFLAGS_HEPMC
      os_data%ldflags_hoppet = DEFAULT_LDFLAGS_HOPPET
      os_data%shlib_ext    = DEFAULT_SHLIB_EXT
      os_data%makeflags    = DEFAULT_MAKEFLAGS
      os_data%prefix       = PREFIX
      os_data%exec_prefix = EXEC_PREFIX
      os_data%bindir       = BINDIR
      os_data%libdir       = LIBDIR
      os_data%includedir  = INCLUDEDIR
      os_data%datarootdir = DATAROOTDIR
      if (present (paths)) then
         if (paths%prefix      /= "")  os_data%prefix       = paths%prefix
         if (paths%exec_prefix /= "")  os_data%exec_prefix = paths%exec_prefix
         if (paths%bindir      /= "")  os_data%bindir       = paths%bindir
         if (paths%libdir      /= "")  os_data%libdir       = paths%libdir
         if (paths%includedir  /= "")  os_data%includedir  = paths%includedir
         if (paths%datarootdir /= "")  os_data%datarootdir = paths%datarootdir
      end if
      if (os_data%use_testfiles) then
         os_data%whizard_omega_binpath  = WHIZARD_TEST_OMEGA_BINPATH
         os_data%whizard_includes       = WHIZARD_TEST_INCLUDES
         os_data%whizard_ldflags        = WHIZARD_TEST_LDFLAGS
         os_data%whizard_libtool        = WHIZARD_LIBTOOL_TEST
         os_data%whizard_modelpath      = WHIZARD_TEST_MODELPATH
         os_data%whizard_models_libpath = WHIZARD_TEST_MODELS_LIBPATH
         os_data%whizard_susypath       = WHIZARD_TEST_SUSYPATH
         os_data%whizard_gmlpath        = WHIZARD_TEST_GMLPATH
         os_data%whizard_cutspath       = WHIZARD_TEST_CUTSPATH
         os_data%whizard_texpath        = WHIZARD_TEST_TEXPATH
         os_data%whizard_testdatapath   = WHIZARD_TEST_TESTDATAPATH
```

```
        os_data%whizard_circe2path      = WHIZARD_TEST_CIRCE2PATH
        os_data%whizard_beamsimpath     = WHIZARD_TEST_BEAMSIMPATH
        os_data%whizard_mulipath        = WHIZARD_TEST_MULIPATH
        os_data%pdf_builtin_datapath    = PDF_BUILTIN_TEST_DATAPATH
     else
        if (os_dir_exist (local_includes)) then
           os_data%whizard_includes = "-I" // local_includes // " "// &
              WHIZARD_INCLUDES
        else
           os_data%whizard_includes = WHIZARD_INCLUDES
        end if
        os_data%whizard_omega_binpath  = WHIZARD_OMEGA_BINPATH
        os_data%whizard_ldflags        = WHIZARD_LDFLAGS
        os_data%whizard_libtool        = WHIZARD_LIBTOOL
        if(present(paths)) then
           if (paths%libtool /= "")  os_data%whizard_libtool = paths%libtool
        end if
        os_data%whizard_modelpath       = WHIZARD_MODELPATH
        os_data%whizard_models_libpath = WHIZARD_MODELS_LIBPATH
        os_data%whizard_susypath        = WHIZARD_SUSYPATH
        os_data%whizard_gmlpath         = WHIZARD_GMLPATH
        os_data%whizard_cutspath        = WHIZARD_CUTSPATH
        os_data%whizard_texpath         = WHIZARD_TEXPATH
        os_data%whizard_testdatapath    = WHIZARD_TESTDATAPATH
        os_data%whizard_circe2path      = WHIZARD_CIRCE2PATH
        os_data%whizard_beamsimpath     = WHIZARD_BEAMSIMPATH
        os_data%whizard_mulipath        = WHIZARD_MULIPATH
        os_data%pdf_builtin_datapath    = PDF_BUILTIN_DATAPATH
     end if
     os_data%event_analysis     = EVENT_ANALYSIS      == "yes"
     os_data%event_analysis_ps  = EVENT_ANALYSIS_PS   == "yes"
     os_data%event_analysis_pdf = EVENT_ANALYSIS_PDF == "yes"
     os_data%latex  = PRG_LATEX // " " // OPT_LATEX
     os_data%mpost  = PRG_MPOST // " " // OPT_MPOST
     os_data%gml    = os_data%whizard_gmlpath // "/gml" // " " // OPT_MPOST &
          // " " // "--gmldir " // os_data%whizard_gmlpath
     os_data%dvips  = PRG_DVIPS
     os_data%ps2pdf = PRG_PS2PDF
     call os_data_expand_paths (os_data)
   end subroutine os_data_init
```

Replace occurences of GNU path variables (such as `${prefix}`) by their values.
Do this for all strings that could depend on them, and do the replacement in
reverse order, since the path variables may be defined in terms of each other.

⟨*OS interface: procedures*⟩+≡

```
   subroutine os_data_expand_paths (os_data)
     type(os_data_t), intent(inout) :: os_data
     integer, parameter :: N_VARIABLES = 6
     type(string_t), dimension(N_VARIABLES) :: variable, value
     variable(1) = "${prefix}";       value(1) = os_data%prefix
     variable(2) = "${exec_prefix}";  value(2) = os_data%exec_prefix
     variable(3) = "${bindir}";       value(3) = os_data%bindir
     variable(4) = "${libdir}";       value(4) = os_data%libdir
```

```
      variable(5) = "${includedir}";    value(5) = os_data%includedir
      variable(6) = "${datarootdir}";  value(6) = os_data%datarootdir
      call expand_paths (os_data%whizard_omega_binpath)
      call expand_paths (os_data%whizard_includes)
      call expand_paths (os_data%whizard_ldflags)
      call expand_paths (os_data%whizard_libtool)
      call expand_paths (os_data%whizard_modelpath)
      call expand_paths (os_data%whizard_models_libpath)
      call expand_paths (os_data%whizard_susypath)
      call expand_paths (os_data%whizard_gmlpath)
      call expand_paths (os_data%whizard_cutspath)
      call expand_paths (os_data%whizard_texpath)
      call expand_paths (os_data%whizard_testdatapath)
      call expand_paths (os_data%whizard_circe2path)
      call expand_paths (os_data%whizard_beamsimpath)
      call expand_paths (os_data%whizard_mulipath)
      call expand_paths (os_data%whizard_models_libpath_local)
      call expand_paths (os_data%whizard_modelpath_local)
      call expand_paths (os_data%whizard_omega_binpath_local)
      call expand_paths (os_data%pdf_builtin_datapath)
      call expand_paths (os_data%latex)
      call expand_paths (os_data%mpost)
      call expand_paths (os_data%gml)
      call expand_paths (os_data%dvips)
      call expand_paths (os_data%ps2pdf)
    contains
      subroutine expand_paths (string)
        type(string_t), intent(inout) :: string
        integer :: i
        do i = N_VARIABLES, 1, -1
           string = replace (string, variable(i), value(i), every=.true.)
        end do
      end subroutine expand_paths
    end subroutine os_data_expand_paths
```

Write contents

⟨*OS interface: public*⟩+≡
```
    public :: os_data_write
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_data_write (os_data, unit)
      type(os_data_t), intent(in) :: os_data
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "OS data:"
      write (u, *) "use_libtool    = ", os_data%use_libtool
      write (u, *) "use_testfiles  = ", os_data%use_testfiles
      write (u, *) "fc             = ", char (os_data%fc)
      write (u, *) "fcflags        = ", char (os_data%fcflags)
      write (u, *) "fcflags_pic    = ", char (os_data%fcflags_pic)
      write (u, *) "fc_src_ext     = ", char (os_data%fc_src_ext)
      write (u, *) "cc             = ", char (os_data%cc)
      write (u, *) "cflags         = ", char (os_data%cflags)
```

```
      write (u, *) "cflags_pic    = ", char (os_data%cflags_pic)
      write (u, *) "obj_ext       = ", char (os_data%obj_ext)
      write (u, *) "ld            = ", char (os_data%ld)
      write (u, *) "ldflags       = ", char (os_data%ldflags)
      write (u, *) "ldflags_so    = ", char (os_data%ldflags_so)
      write (u, *) "ldflags_static = ", char (os_data%ldflags_static)
      write (u, *) "ldflags_hepmc = ", char (os_data%ldflags_hepmc)
      write (u, *) "ldflags_hoppet = ", char (os_data%ldflags_hoppet)
      write (u, *) "shlib_ext     = ", char (os_data%shlib_ext)
      write (u, *) "makeflags     = ", char (os_data%makeflags)
      write (u, *) "prefix        = ", char (os_data%prefix)
      write (u, *) "exec_prefix   = ", char (os_data%exec_prefix)
      write (u, *) "bindir        = ", char (os_data%bindir)
      write (u, *) "libdir        = ", char (os_data%libdir)
      write (u, *) "includedir    = ", char (os_data%includedir)
      write (u, *) "datarootdir   = ", char (os_data%datarootdir)
      write (u, *) "whizard_omega_binpath = ", &
           char (os_data%whizard_omega_binpath)
      write (u, *) "whizard_includes      = ", char (os_data%whizard_includes)
      write (u, *) "whizard_ldflags       = ", char (os_data%whizard_ldflags)
      write (u, *) "whizard_libtool       = ", char (os_data%whizard_libtool)
      write (u, *) "whizard_modelpath     = ", &
           char (os_data%whizard_modelpath)
      write (u, *) "whizard_models_libpath = ", &
           char (os_data%whizard_modelpath)
      write (u, *) "whizard_susypath      = ", char (os_data%whizard_susypath)
      write (u, *) "whizard_gmlpath       = ", char (os_data%whizard_gmlpath)
      write (u, *) "whizard_cutspath      = ", char (os_data%whizard_cutspath)
      write (u, *) "whizard_texpath       = ", char (os_data%whizard_texpath)
      write (u, *) "whizard_circe2path    = ", char (os_data%whizard_circe2path)
      write (u, *) "whizard_beamsimpath   = ", char (os_data%whizard_beamsimpath)
      write (u, *) "whizard_mulipath  = ", char (os_data%whizard_mulipath)
      write (u, *) "whizard_testdatapath  = ", &
           char (os_data%whizard_testdatapath)
      write (u, *) "whizard_modelpath_local    = ", &
           char (os_data%whizard_modelpath_local)
      write (u, *) "whizard_models_libpath_local = ", &
           char (os_data%whizard_models_libpath_local)
      write (u, *) "whizard_omega_binpath_local = ", &
           char (os_data%whizard_omega_binpath_local)
      write (u, *) "event_analysis     = ", os_data%event_analysis
      write (u, *) "event_analysis_ps  = ", os_data%event_analysis_ps
      write (u, *) "event_analysis_pdf = ", os_data%event_analysis_pdf
      write (u, *) "latex  = ", char (os_data%latex)
      write (u, *) "mpost  = ", char (os_data%mpost)
      write (u, *) "gml    = ", char (os_data%gml)
      write (u, *) "dvips  = ", char (os_data%dvips)
      write (u, *) "ps2pdf = ", char (os_data%ps2pdf)
    end subroutine os_data_write
```

2.5.3 Dynamic linking

We define a type that holds the filehandle for a dynamically linked library (shared object), together with functions to open and close the library, and to access functions in this library.

⟨*OS interface: public*⟩+≡
```
    public :: dlaccess_t
```

⟨*OS interface: types*⟩+≡
```
    type :: dlaccess_t
       private
       type(string_t) :: filename
       type(c_ptr) :: handle = c_null_ptr
       logical :: is_open = .false.
       logical :: has_error = .false.
       type(string_t) :: error
     contains
     ⟨OS interface: dlaccess: TBP⟩
    end type dlaccess_t
```

Output. This is called by the output routine for the process library.

⟨*OS interface: dlaccess: TBP*⟩≡
```
    procedure :: write => dlaccess_write
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine dlaccess_write (object, unit)
      class(dlaccess_t), intent(in) :: object
      integer, intent(in) :: unit
      write (unit, "(1x,A)")  "DL access info:"
      write (unit, "(3x,A,L1)")  "is open  = ", object%is_open
      if (object%has_error) then
         write (unit, "(3x,A,A,A)")  "error    = '", char (object%error), "'"
      else
         write (unit, "(3x,A)")      "error    = [none]"
      end if
    end subroutine dlaccess_write
```

The interface to the library functions:

⟨*OS interface: interfaces*⟩≡
```
    interface
       function dlopen (filename, flag) result (handle) bind(C)
         import
         character(c_char), dimension(*) :: filename
         integer(c_int), value :: flag
         type(c_ptr) :: handle
       end function dlopen
    end interface

    interface
       function dlclose (handle) result (status) bind(C)
         import
         type(c_ptr), value :: handle
         integer(c_int) :: status
       end function dlclose
```

```
      end interface

      interface
         function dlerror () result (str) bind(C)
            import
            type(c_ptr) :: str
         end function dlerror
      end interface

      interface
         function dlsym (handle, symbol) result (fptr) bind(C)
            import
            type(c_ptr), value :: handle
            character(c_char), dimension(*) :: symbol
            type(c_funptr) :: fptr
         end function dlsym
      end interface
```

This reads an error string and transforms it into a `string_t` object, if an error
has occured. If not, set the error flag to false and return an empty string.

⟨*Limits: public parameters*⟩+≡
```
      integer, parameter, public :: DLERROR_LEN = 160
```

⟨*OS interface: procedures*⟩+≡
```
      subroutine read_dlerror (has_error, error)
        logical, intent(out) :: has_error
        type(string_t), intent(out) :: error
        type(c_ptr) :: err_cptr
        character(len=DLERROR_LEN, kind=c_char), pointer :: err_fptr
        integer :: str_end
        err_cptr = dlerror ()
        if (c_associated (err_cptr)) then
           call c_f_pointer (err_cptr, err_fptr)
           has_error = .true.
           str_end = scan (err_fptr, c_null_char)
           if (str_end > 0) then
              error = err_fptr(1:str_end-1)
           else
              error = err_fptr
           end if
        else
           has_error = .false.
           error = ""
        end if
      end subroutine read_dlerror
```

This is the Fortran API. Init/final open and close the file, i.e., load and unload
the library.

Note that a library can be opened more than once, and that for an ultimate
close as many `dlclose` calls as `dlopen` calls are necessary. However, we assume
that it is opened and closed only once.

⟨*OS interface: public*⟩+≡
```
      public :: dlaccess_init
```

```
    public :: dlaccess_final
```
⟨*OS interface: dlaccess: TBP*⟩+≡
```
    procedure :: init => dlaccess_init
    procedure :: final => dlaccess_final
```
⟨*OS interface: procedures*⟩+≡
```
    subroutine dlaccess_init (dlaccess, prefix, libname, os_data)
      class(dlaccess_t), intent(out) :: dlaccess
      type(string_t), intent(in) :: prefix, libname
      type(os_data_t), intent(in), optional :: os_data
      type(string_t) :: filename
      logical :: exist
      dlaccess%filename = libname
      filename = prefix // "/" // libname
      inquire (file=char(filename), exist=exist)
      if (.not. exist) then
          filename = prefix // "/.libs/" // libname
          inquire (file=char(filename), exist=exist)
          if (.not. exist) then
             dlaccess%has_error = .true.
             dlaccess%error = "Library '" // filename // "' not found"
             return
          end if
      end if
      dlaccess%handle = dlopen (char (filename) // c_null_char, ior ( &
          RTLD_LAZY, RTLD_LOCAL))
      dlaccess%is_open = c_associated (dlaccess%handle)
      call read_dlerror (dlaccess%has_error, dlaccess%error)
    end subroutine dlaccess_init

    subroutine dlaccess_final (dlaccess)
      class(dlaccess_t), intent(inout) :: dlaccess
      integer(c_int) :: status
      if (dlaccess%is_open) then
          status = dlclose (dlaccess%handle)
          dlaccess%is_open = .false.
          call read_dlerror (dlaccess%has_error, dlaccess%error)
      end if
    end subroutine dlaccess_final
```

Return true if an error has occured.

⟨*OS interface: public*⟩+≡
```
    public :: dlaccess_has_error
```
⟨*OS interface: procedures*⟩+≡
```
    function dlaccess_has_error (dlaccess) result (flag)
      logical :: flag
      type(dlaccess_t), intent(in) :: dlaccess
      flag = dlaccess%has_error
    end function dlaccess_has_error
```

Return the error string currently stored in the `dlaccess` object.

⟨*OS interface: public*⟩+≡
```
    public :: dlaccess_get_error
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_get_error (dlaccess) result (error)
  type(string_t) :: error
  type(dlaccess_t), intent(in) :: dlaccess
  error = dlaccess%error
end function dlaccess_get_error
```

The symbol handler returns the C address of the function with the given string name. (It is a good idea to use `bind(C)` for all functions accessed by this, such that the name string is well-defined.) Call `c_f_procpointer` to cast this into a Fortran procedure pointer with an appropriate interface.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_get_c_funptr
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_get_c_funptr (dlaccess, fname) result (fptr)
  type(c_funptr) :: fptr
  type(dlaccess_t), intent(inout) :: dlaccess
  type(string_t), intent(in) :: fname
  fptr = dlsym (dlaccess%handle, char (fname) // c_null_char)
  call read_dlerror (dlaccess%has_error, dlaccess%error)
end function dlaccess_get_c_funptr
```

2.5.4 Predicates

Return true if the library is loaded. In particular, this is false if loading was unsuccessful.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_is_open
```

⟨*OS interface: procedures*⟩+≡
```
function dlaccess_is_open (dlaccess) result (flag)
  logical :: flag
  type(dlaccess_t), intent(in) :: dlaccess
  flag = dlaccess%is_open
end function dlaccess_is_open
```

2.5.5 Shell access

This is the standard system call for executing a shell command, such as invoking a compiler.

In F2008 there will be the equivalent built-in command `execute_command_line`.

⟨*OS interface: public*⟩+≡
```
public :: os_system_call
```

⟨*OS interface: procedures*⟩+≡
```
subroutine os_system_call (command_string, status, verbose)
  type(string_t), intent(in) :: command_string
  integer, intent(out), optional :: status
  logical, intent(in), optional :: verbose
  logical :: verb
```

```
      integer :: stat
      verb = .false.;  if (present (verbose))  verb = verbose
      if (verb) &
          call msg_message ("command: " // char (command_string))
      stat = system (char (command_string) // c_null_char)
      if (present (status)) then
         status = stat
      else if (stat /= 0) then
         if (.not. verb) &
             call msg_message ("command: " // char (command_string))
         write (msg_buffer, "(A,I0)")  "Return code = ", stat
         call msg_message ()
         call msg_fatal ("System command returned with nonzero status code")
      end if
   end subroutine os_system_call
```

⟨*OS interface: interfaces*⟩+≡
```
   interface
     function system (command) result (status) bind(C)
       import
       integer(c_int) :: status
       character(c_char), dimension(*) :: command
     end function system
   end interface
```

2.5.6   Querying for a directory

This queries for the existence of a directory. There is no standard way to achieve this in FORTRAN, and if we were to call into `libc`, we would need access to C macros for evaluating the result, so we resort to calling `test` as a system call.

⟨*OS interface: public*⟩+≡
```
   public :: os_dir_exist
```

⟨*OS interface: procedures*⟩+≡
```
   function os_dir_exist (name) result (res)
     type(string_t), intent(in) :: name
     logical :: res
     integer :: status
     call os_system_call ('test -d "' // name // '"', status=status)
     res = status == 0
   end function os_dir_exist
```

2.5.7   Fortran compiler and linker

Compile a single module for use in a shared library, but without linking.

⟨*OS interface: public*⟩+≡
```
   public :: os_compile_shared
```

⟨*OS interface: procedures*⟩+≡
```
   subroutine os_compile_shared (src, os_data, status)
     type(string_t), intent(in) :: src
     type(os_data_t), intent(in) :: os_data
```

```
   integer, intent(out), optional :: status
   type(string_t) :: command_string
   if (os_data%use_libtool) then
      command_string = &
           os_data%whizard_libtool // " --mode=compile " // &
           os_data%fc // " " // &
           "-c " // &
           os_data%whizard_includes // " " // &
           os_data%fcflags // " " // &
           "'" // src // os_data%fc_src_ext // "'"
   else
      command_string = &
           os_data%fc // " " // &
           "-c  " // &
           os_data%fcflags_pic // " " // &
           os_data%whizard_includes // " " // &
           os_data%fcflags // " " // &
           "'" // src // os_data%fc_src_ext // "'"
   end if
   call os_system_call (command_string, status)
  end subroutine os_compile_shared
```

Link an array of object files to build a shared object library. In the libtool
case, we have to specify a -rpath, otherwise only a static library can be built.
However, since the library is never installed, this rpath is irrelevant.

⟨*OS interface: public*⟩+≡
```
   public :: os_link_shared
```

⟨*OS interface: procedures*⟩+≡
```
  subroutine os_link_shared (objlist, lib, os_data, status)
    type(string_t), intent(in) :: objlist, lib
    type(os_data_t), intent(in) :: os_data
    integer, intent(out), optional :: status
    type(string_t) :: command_string
    if (os_data%use_libtool) then
       command_string = &
            os_data%whizard_libtool // " --mode=link " // &
            os_data%fc // " " // &
            "-module " // &
            "-rpath /usr/local/lib" // " " // &
            os_data%fcflags // " " // &
            os_data%whizard_ldflags // " " // &
            os_data%ldflags // " " // &
            "-o '" // lib // ".la' " // &
            objlist
    else
       command_string = &
            os_data%ld // " " // &
            os_data%ldflags_so // " " // &
            os_data%fcflags // " " // &
            os_data%whizard_ldflags // " " // &
            os_data%ldflags // " " // &
            "-o '" // lib // os_data%shlib_ext // "' " // &
            objlist
```

46

```
      end if
      call os_system_call (command_string, status)
    end subroutine os_link_shared
```

Link an array of object files / libraries to build a static executable.

⟨*OS interface: public*⟩+≡
```
    public :: os_link_static
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_link_static (objlist, exec_name, os_data, status)
      type(string_t), intent(in) :: objlist, exec_name
      type(os_data_t), intent(in) :: os_data
      integer, intent(out), optional :: status
      type(string_t) :: command_string
      if (os_data%use_libtool) then
         command_string = &
              os_data%whizard_libtool // " --mode=link " // &
              os_data%fc // " " // &
              "-static-libtool-libs " // &
              os_data%fcflags // " " // &
              os_data%whizard_ldflags // " " // &
              os_data%ldflags // " " // &
              os_data%ldflags_static // " " // &
              "-o '" // exec_name // "' " // &
              objlist // " " // &
              os_data%ldflags_hepmc // " " // &
              os_data%ldflags_hoppet
      else
         command_string = &
              os_data%ld // " " // &
              os_data%ldflags_so // " " // &
              os_data%fcflags // " " // &
              os_data%whizard_ldflags // " " // &
              os_data%ldflags // " " // &
              os_data%ldflags_static // " " // &
              "-o '" // exec_name // "' " // &
              objlist // " " // &
              os_data%ldflags_hepmc // " " // &
              os_data%ldflags_hoppet
      end if
      call os_system_call (command_string, status)
    end subroutine os_link_static
```

Determine the name of the shared library to link. If libtool is used, this is encoded in the `.la` file which resides in place of the library itself.

⟨*OS interface: public*⟩+≡
```
    public :: os_get_dlname
```

⟨*OS interface: procedures*⟩+≡
```
    function os_get_dlname (lib, os_data, ignore, silent) result (dlname)
      type(string_t) :: dlname
      type(string_t), intent(in) :: lib
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: ignore, silent
```

```
   type(string_t) :: filename
   type(string_t) :: buffer
   logical :: exist, required, quiet
   integer :: u
   u = free_unit ()
   if (present (ignore)) then
      required = .not. ignore
   else
      required = .true.
   end if
      if (present (silent)) then
      quiet = silent
   else
      quiet = .false.
   end if
   if (os_data%use_libtool) then
      filename = lib // ".la"
      inquire (file=char(filename), exist=exist)
      if (exist) then
         open (unit=u, file=char(filename), action="read", status="old")
         SCAN_LTFILE: do
            call get (u, buffer)
            if (extract (buffer, 1, 7) == "dlname=") then
               dlname = extract (buffer, 9)
               dlname = remove (dlname, len (dlname))
               exit SCAN_LTFILE
            end if
         end do SCAN_LTFILE
         close (u)
      else if (required) then
         if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
            // "': libtool archive not found")
         dlname = ""
      else
         if (.not. quiet) call msg_message ("[No compiled library '" &
            // char (lib) // "']")
         dlname = ""
      end if
   else
      dlname = lib // os_data%shlib_ext
      inquire (file=char(dlname), exist=exist)
      if (.not. exist) then
         if (required) then
            if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
               // "' not found")
         else
            if (.not. quiet) call msg_message &
               ("[No compiled process library '" // char (lib) // "']")
            dlname = ""
         end if
      end if
   end if
end function os_get_dlname
```

2.5.8 Controlling OpenMP

OpenMP is handled automatically by the library for the most part. Here is a convenience routine for setting the number of threads, with some diagnostics.

⟨*OS interface: public*⟩+≡
```
    public :: openmp_set_num_threads_verbose
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine openmp_set_num_threads_verbose (num_threads, openmp_logging)
      integer, intent(in) :: num_threads
      integer :: n_threads
      logical, intent(in), optional :: openmp_logging
      logical :: logging
      if (present (openmp_logging)) then
         logging = openmp_logging
      else
         logging = .true.
      end if
      n_threads = num_threads
      if (openmp_is_active ()) then
         if (num_threads == 1) then
            if (logging) then
               write (msg_buffer, "(A,I0,A)")  "OpenMP: Using ", num_threads, &
                   " thread"
               call msg_message
            end if
            n_threads = num_threads
         else if (num_threads > 1) then
            if (logging) then
               write (msg_buffer, "(A,I0,A)")  "OpenMP: Using ", num_threads, &
                   " threads"
               call msg_message
            end if
            n_threads = num_threads
         else
            if (logging) then
               write (msg_buffer, "(A,I0,A)")  "OpenMP: " &
                   // "Illegal value of openmp_num_threads (", num_threads, &
                 ") ignored"
               call msg_error
            end if
            n_threads = openmp_get_default_max_threads ()
            if (logging) then
               write (msg_buffer, "(A,I0,A)")  "OpenMP: Using ", &
                   n_threads, " threads"
               call msg_message
            end if
         end if
         if (n_threads > openmp_get_default_max_threads ()) then
            if (logging) then
               write (msg_buffer, "(A,I0)")  "OpenMP: " &
                   // "Number of threads is greater than library default of ", &
                   openmp_get_default_max_threads ()
               call msg_warning
            end if
```

```
         end if
         call openmp_set_num_threads (n_threads)
      else if (num_threads /= 1) then
         if (logging) then
            write (msg_buffer, "(A,I0,A)")  "openmp_num_threads set to ", &
                 num_threads, ", but OpenMP is not active: ignored"
            call msg_warning
         end if
      end if
   end subroutine openmp_set_num_threads_verbose
```

2.5.9   Unit tests

⟨*OS interface: public*⟩+≡
```
    public :: os_interface_test
```

⟨*OS interface: procedures*⟩+≡
```
    subroutine os_interface_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*OS interface: execute tests*⟩
```
    end subroutine os_interface_test
```

Write a Fortran source file, compile it to a shared library, load it, and execute
the contained function.

⟨*OS interface: execute tests*⟩≡
```
    call test (os_interface_1, "os_interface_1", &
         "check OS interface routines", &
         u, results)
```

⟨*OS interface: tests*⟩≡
```
    subroutine os_interface_1 (u)
      integer, intent(in) :: u
      type(dlaccess_t) :: dlaccess
      type(string_t) :: fname, libname, ext
      type(os_data_t) :: os_data
      type(string_t) :: filename_src, filename_obj
      abstract interface
         function so_test_proc (i) result (j) bind(C)
           import c_int
           integer(c_int), intent(in) :: i
           integer(c_int) :: j
         end function so_test_proc
      end interface
      procedure(so_test_proc), pointer :: so_test => null ()
      type(c_funptr) :: c_fptr
      integer :: unit
      integer(c_int) :: i
      call os_data_init (os_data)
      fname = "so_test"
      filename_src = fname // os_data%fc_src_ext
      if (os_data%use_libtool) then
         ext = ".lo"
```

```
else
   ext = os_data%obj_ext
end if
filename_obj = fname // ext
libname = fname // os_data%shlib_ext

write (u, "(A)")  "* Test output: OS interface"
write (u, "(A)")  "*   Purpose: check os_interface routines"
write (u, "(A)")

write (u, "(A)")  "* write source file 'so_test.f90'"
write (u, "(A)")
unit = free_unit ()
open (unit=unit, file=char(filename_src), action="write")
write (unit, "(A)")  "function so_test (i) result (j) bind(C)"
write (unit, "(A)")  "  use iso_c_binding"
write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
write (unit, "(A)")  "  integer(c_int) :: j"
write (unit, "(A)")  "  j = 2 * i"
write (unit, "(A)")  "end function so_test"
close (unit)
write (u, "(A)")  "* compile and link as 'so_test.so/dylib'"
write (u, "(A)")
call os_compile_shared (fname, os_data)
call os_link_shared (filename_obj, fname, os_data)
write (u, "(A)")  "* load library 'so_test.so/dylib'"
write (u, "(A)")
call dlaccess_init (dlaccess, var_str ("."), libname, os_data)
if (dlaccess_is_open (dlaccess)) then
   write (u, "(A)") "    success"
else
   write (u, "(A)") "    failure"
end if
write (u, "(A)")  "* load symbol 'so_test'"
write (u, "(A)")
c_fptr = dlaccess_get_c_funptr (dlaccess, fname)
if (c_associated (c_fptr)) then
   write (u, "(A)") "    success"
else
   write (u, "(A)") "    failure"
end if
call c_f_procpointer (c_fptr, so_test)
write (u, "(A)") "* Execute function from 'so_test.so/dylib'"
i = 7
write (u, "(A,1x,I1)") "    input  = ", i
write (u, "(A,1x,I1)") "    result = ", so_test(i)
if (so_test(i) / i .ne. 2) then
   write (u, "(A)")  "* Compiling and linking ISO C functions failed."
else
   write (u, "(A)")  "* Successful."
end if
write (u, "(A)")
write (u, "(A)")  "* Cleanup"
call dlaccess_final (dlaccess)
```

```
      end subroutine os_interface_1
```

2.6   CPU timing

This is a simplified module which replaces `clock.f90` below. The time is now stored in a simple derived type which just holds a floating-point number.

⟨`cputime.f90`⟩≡
  ⟨*File header*⟩

```
  module cputime
```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
```
    use diagnostics !NODEP!
    use unit_tests
```

  ⟨*Standard module head*⟩

  ⟨*CPU time: public*⟩

  ⟨*CPU time: types*⟩

  ⟨*CPU time: interfaces*⟩

```
  contains
```

  ⟨*CPU time: procedures*⟩

  ⟨*CPU time: tests*⟩

```
  end module cputime
```

The CPU time is a floating-point number with an arbitrary reference time. It is single precision (default real, not `real(default)`). It is measured in seconds.

⟨*CPU time: public*⟩≡
```
    public :: time_t
```

⟨*CPU time: types*⟩≡
```
    type :: time_t
       private
       logical :: known = .false.
       real :: value = 0
     contains
```
    ⟨*CPU time: time: TBP*⟩
```
    end type time_t
```

⟨*CPU time: time: TBP*⟩≡
```
    procedure :: write => time_write
```

⟨*CPU time: procedures*⟩≡
```
    subroutine time_write (object, unit)
      class(time_t), intent(in) :: object
```

```
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "Time in seconds ="
      if (object%known) then
         write (u, "(1x,ES10.3)")  object%value
      else
         write (u, "(1x,A)")  "[unknown]"
      end if
   end subroutine time_write
```

Set the current time

⟨*CPU time: time: TBP*⟩+≡
```
   procedure :: set_current => time_set_current
```

⟨*CPU time: procedures*⟩+≡
```
   subroutine time_set_current (time)
     class(time_t), intent(out) :: time
     integer :: msecs
     call system_clock (msecs)
     time%value = real (msecs) / 1000.
     time%known = time%value > 0
   end subroutine time_set_current
```

Assign to a `real(default` value. If the time is undefined, return zero.

⟨*CPU time: public*⟩+≡
```
   public :: assignment(=)
```

⟨*CPU time: interfaces*⟩≡
```
   interface assignment(=)
     module procedure real_assign_time
     module procedure real_default_assign_time
   end interface
```

⟨*CPU time: procedures*⟩+≡
```
   pure subroutine real_assign_time (r, time)
     real, intent(out) :: r
     class(time_t), intent(in) :: time
     if (time%known) then
        r = time%value
     else
        r = 0
     end if
   end subroutine real_assign_time

   pure subroutine real_default_assign_time (r, time)
     real(default), intent(out) :: r
     class(time_t), intent(in) :: time
     if (time%known) then
        r = time%value
     else
        r = 0
     end if
```

```
      end subroutine real_default_assign_time
```

Assign an integer or (single precision) real value to the time object.

⟨*CPU time: time: TBP*⟩+≡
```
    generic :: assignment(=) => time_assign_from_integer, time_assign_from_real
    procedure, private :: time_assign_from_integer
    procedure, private :: time_assign_from_real
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine time_assign_from_integer (time, ival)
      class(time_t), intent(out) :: time
      integer, intent(in) :: ival
      time%value = ival
      time%known = .true.
    end subroutine time_assign_from_integer

    subroutine time_assign_from_real (time, rval)
      class(time_t), intent(out) :: time
      real, intent(in) :: rval
      time%value = rval
      time%known = .true.
    end subroutine time_assign_from_real
```

Add times and compute time differences. If any input value is undefined, the result is undefined.

⟨*CPU time: time: TBP*⟩+≡
```
    generic :: operator(-) => subtract_times
    generic :: operator(+) => add_times
    procedure, private :: subtract_times
    procedure, private :: add_times
```

⟨*CPU time: procedures*⟩+≡
```
    pure function subtract_times (t_end, t_begin) result (time)
      type(time_t) :: time
      class(time_t), intent(in) :: t_end, t_begin
      if (t_end%known .and. t_begin%known) then
         time%known = .true.
         time%value = t_end%value - t_begin%value
      end if
    end function subtract_times

    pure function add_times (t1, t2) result (time)
      type(time_t) :: time
      class(time_t), intent(in) :: t1, t2
      if (t1%known .and. t2%known) then
         time%known = .true.
         time%value = t1%value + t2%value
      end if
    end function add_times
```

Check if a time is known, so we can use it:

⟨*CPU time: time: TBP*⟩+≡
```
    procedure :: is_known => time_is_known
```

⟨*CPU time: procedures*⟩+≡
```
    function time_is_known (time) result (flag)
      class(time_t), intent(in) :: time
      logical :: flag
      flag = time%known
    end function time_is_known
```

We define functions for converting the time into ss / mm:ss / hh:mm:ss / dd:mm:hh:ss.

⟨*CPU time: time: TBP*⟩+≡
```
    generic :: expand => time_expand_s, time_expand_ms, &
        time_expand_hms, time_expand_dhms
    procedure, private :: time_expand_s
    procedure, private :: time_expand_ms
    procedure, private :: time_expand_hms
    procedure, private :: time_expand_dhms
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine time_expand_s (time, sec)
      class(time_t), intent(in) :: time
      integer, intent(out) :: sec
      if (time%known) then
         sec = time%value
      else
         call msg_bug ("Time: attempt to expand undefined value")
      end if
    end subroutine time_expand_s

    subroutine time_expand_ms (time, min, sec)
      class(time_t), intent(in) :: time
      integer, intent(out) :: min, sec
      if (time%known) then
         if (time%value >= 0) then
            sec = mod (int (time%value), 60)
         else
            sec = - mod (int (- time%value), 60)
         end if
         min = time%value / 60
      else
         call msg_bug ("Time: attempt to expand undefined value")
      end if
    end subroutine time_expand_ms

    subroutine time_expand_hms (time, hour, min, sec)
      class(time_t), intent(in) :: time
      integer, intent(out) :: hour, min, sec
      call time%expand (min, sec)
      hour = min / 60
      if (min >= 0) then
         min = mod (min, 60)
      else
         min = - mod (-min, 60)
      end if
    end subroutine time_expand_hms
```

```
subroutine time_expand_dhms (time, day, hour, min, sec)
  class(time_t), intent(in) :: time
  integer, intent(out) :: day, hour, min, sec
  call time%expand (hour, min, sec)
  day = hour / 24
  if (hour >= 0) then
     hour = mod (hour, 24)
  else
     hour = - mod (- hour, 24)
  end if
end subroutine time_expand_dhms
```

Use the above expansions to generate a time string.

⟨*CPU time: time: TBP*⟩+≡
```
procedure :: to_string_s => time_to_string_s
procedure :: to_string_ms => time_to_string_ms
procedure :: to_string_hms => time_to_string_hms
procedure :: to_string_dhms => time_to_string_dhms
```

⟨*CPU time: procedures*⟩+≡
```
function time_to_string_s (time) result (str)
  class(time_t), intent(in) :: time
  type(string_t) :: str
  character(256) :: buffer
  integer :: s
  call time%expand (s)
  write (buffer, "(I0,'s')")  s
  str = trim (buffer)
end function time_to_string_s

function time_to_string_ms (time, blank) result (str)
  class(time_t), intent(in) :: time
  logical, intent(in), optional :: blank
  type(string_t) :: str
  character(256) :: buffer
  integer :: s, m
  logical :: x_out
  x_out = .false.
  if (present (blank))  x_out = blank
  call time%expand (m, s)
  write (buffer, "(I0,'m:',I2.2,'s')")  m, abs (s)
  str = trim (buffer)
  if (x_out) then
     str = replace (str, len(str)-1, "X")
  end if
end function time_to_string_ms

function time_to_string_hms (time) result (str)
  class(time_t), intent(in) :: time
  type(string_t) :: str
  character(256) :: buffer
  integer :: s, m, h
  call time%expand (h, m, s)
```

```
      write (buffer, "(I0,'h:',I2.2,'m:',I2.2,'s')")  h, abs (m), abs (s)
      str = trim (buffer)
    end function time_to_string_hms

    function time_to_string_dhms (time) result (str)
      class(time_t), intent(in) :: time
      type(string_t) :: str
      character(256) :: buffer
      integer :: s, m, h, d
      call time%expand (d, h, m, s)
      write (buffer, "(I0,'d:',I2.2,'h:',I2.2,'m:',I2.2,'s')")  &
            d, abs (h), abs (m), abs (s)
      str = trim (buffer)
    end function time_to_string_dhms
```

2.6.1 Timer

A timer can measure real (wallclock) time differences. The base type corresponds to the result, i.e., time difference. The object contains two further times for start and stop time.

⟨*CPU time: public*⟩+≡
```
    public :: timer_t
```

⟨*CPU time: types*⟩+≡
```
    type, extends (time_t) :: timer_t
      private
      logical :: running = .false.
      type(time_t) :: t1, t2
     contains
     ⟨CPU time: timer: TBP⟩
    end type timer_t
```

Output. If the timer is running, we indicate this, otherwise write just the result.

⟨*CPU time: timer: TBP*⟩≡
```
    procedure :: write => timer_write
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine timer_write (object, unit)
      class(timer_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (object%running) then
         write (u, "(1x,A)")  "Time in seconds = [running]"
      else
         call object%time_t%write (u)
      end if
    end subroutine timer_write
```

Start the timer: store the current time in the first entry and adapt the status. We forget any previous values.

⟨*CPU time: timer: TBP*⟩+≡

```
    procedure :: start => timer_start
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine timer_start (timer)
      class(timer_t), intent(out) :: timer
      call timer%t1%set_current ()
      timer%running = .true.
    end subroutine timer_start
```

Restart the timer: simply adapt the status, keeping the start time.

⟨*CPU time: timer: TBP*⟩+≡
```
    procedure :: restart => timer_restart
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine timer_restart (timer)
      class(timer_t), intent(inout) :: timer
      if (timer%t1%known .and. .not. timer%running) then
         timer%running = .true.
      else
         call msg_bug ("Timer: restart attempt from wrong status")
      end if
    end subroutine timer_restart
```

Stop the timer: store the current time in the second entry, adapt the status, and compute the elapsed time.

⟨*CPU time: timer: TBP*⟩+≡
```
    procedure :: stop => timer_stop
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine timer_stop (timer)
      class(timer_t), intent(inout) :: timer
      call timer%t2%set_current ()
      timer%running = .false.
      call timer%evaluate ()
    end subroutine timer_stop
```

This is separate, available for the unit test.

⟨*CPU time: timer: TBP*⟩+≡
```
    procedure, private :: evaluate => timer_evaluate
```

⟨*CPU time: procedures*⟩+≡
```
    subroutine timer_evaluate (timer)
      class(timer_t), intent(inout) :: timer
      timer%time_t = timer%t2 - timer%t1
    end subroutine timer_evaluate
```

2.6.2   Unit tests

⟨*CPU time: public*⟩+≡
```
    public :: cputime_test
```

⟨*CPU time: tests*⟩≡
```
    subroutine cputime_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨CPU time: execute tests⟩
    end subroutine cputime_test
```

Basic tests

Check basic functions of the time object. The part which we can't check is getting the actual time from the system clock, since the output will not be reproducible. However, we can check time formats and operations.

⟨*CPU time: execute tests*⟩≡
```
    call test (cputime_1, "cputime_1", &
         "time operations", &
         u, results)
```

⟨*CPU time: tests*⟩+≡
```
    subroutine cputime_1 (u)
      integer, intent(in) :: u
      type(time_t) :: time, time1, time2
      real :: t
      integer :: d, h, m, s

      write (u, "(A)")  "* Test output: cputime_1"
      write (u, "(A)")  "*   Purpose: check time operations"
      write (u, "(A)")

      write (u, "(A)") "* Undefined time"
      write (u, *)

      call time%write (u)

      write (u, *)
      write (u, "(A)") "* Set time to zero"
      write (u, *)

      time = 0
      call time%write (u)

      write (u, *)
      write (u, "(A)") "* Set time to 1.234 s"
      write (u, *)

      time = 1.234
      call time%write (u)

      t = time
      write (u, "(1x,A,F6.3)")  "Time as real =", t

      write (u, *)
      write (u, "(A)") "* Compute time difference"
      write (u, *)
```

59

```
time1 = 5.33
time2 = 7.55
time = time2 - time1

call time1%write (u)
call time2%write (u)
call time%write (u)

write (u, *)
write (u, "(A)") "* Compute time sum"
write (u, *)

time = time2 + time1

call time1%write (u)
call time2%write (u)
call time%write (u)

write (u, *)
write (u, "(A)") "* Expand time"
write (u, *)

time1 = ((24 + 1) * 60 + 1) * 60 + 1
time2 = ((3 * 24 + 23) * 60 + 59) * 60 + 59

call time1%expand (s)
write (u, 1)  "s =", s
call time1%expand (m,s)
write (u, 1)  "ms =", m, s
call time1%expand (h,m,s)
write (u, 1)  "hms =", h, m, s
call time1%expand (d,h,m,s)
write (u, 1)  "dhms =", d, h, m, s

call time2%expand (s)
write (u, 1)  "s =", s
call time2%expand (m,s)
write (u, 1)  "ms =", m, s
call time2%expand (h,m,s)
write (u, 1)  "hms =", h, m, s
call time2%expand (d,h,m,s)
write (u, 1)  "dhms =", d, h, m, s

write (u, *)
write (u, "(A)") "* Expand negative time"
write (u, *)

time1 = - (((24 + 1) * 60 + 1) * 60 + 1)
time2 = - (((3 * 24 + 23) * 60 + 59) * 60 + 59)

call time1%expand (s)
write (u, 1)  "s =", s
call time1%expand (m,s)
```

```
      write (u, 1)  "ms =", m, s
      call time1%expand (h,m,s)
      write (u, 1)  "hms =", h, m, s
      call time1%expand (d,h,m,s)
      write (u, 1)  "dhms =", d, h, m, s

      call time2%expand (s)
      write (u, 1)  "s =", s
      call time2%expand (m,s)
      write (u, 1)  "ms =", m, s
      call time2%expand (h,m,s)
      write (u, 1)  "hms =", h, m, s
      call time2%expand (d,h,m,s)
      write (u, 1)  "dhms =", d, h, m, s

1     format (1x,A,1x,4(I0,:,':'))

      write (u, *)
      write (u, "(A)") "* String from time"
      write (u, *)

      time1 = ((24 + 1) * 60 + 1) * 60 + 1
      time2 = ((3 * 24 + 23) * 60 + 59) * 60 + 59

      write (u, "(A)")  char (time1%to_string_s ())
      write (u, "(A)")  char (time1%to_string_ms ())
      write (u, "(A)")  char (time1%to_string_hms ())
      write (u, "(A)")  char (time1%to_string_dhms ())

      write (u, "(A)")  char (time2%to_string_s ())
      write (u, "(A)")  char (time2%to_string_ms ())
      write (u, "(A)")  char (time2%to_string_hms ())
      write (u, "(A)")  char (time2%to_string_dhms ())

      write (u, "(A)")
      write (u, "(A)")  "* Blanking out the last second entry"
      write (u, "(A)")

      write (u, "(A)")  char (time1%to_string_ms ())
      write (u, "(A)")  char (time1%to_string_ms (.true.))

      write (u, *)
      write (u, "(A)") "* String from negative time"
      write (u, *)

      time1 = -(((24 + 1) * 60 + 1) * 60 + 1)
      time2 = -(((3 * 24 + 23) * 60 + 59) * 60 + 59)

      write (u, "(A)")  char (time1%to_string_s ())
      write (u, "(A)")  char (time1%to_string_ms ())
      write (u, "(A)")  char (time1%to_string_hms ())
      write (u, "(A)")  char (time1%to_string_dhms ())

      write (u, "(A)")  char (time2%to_string_s ())
```

```
        write (u, "(A)")  char (time2%to_string_ms ())
        write (u, "(A)")  char (time2%to_string_hms ())
        write (u, "(A)")  char (time2%to_string_dhms ())

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: cputime_1"

    end subroutine cputime_1
```

Timer tests

Check a timer object.

⟨*CPU time: execute tests*⟩+≡
```
    call test (cputime_2, "cputime_2", &
         "timer", &
         u, results)
```

⟨*CPU time: tests*⟩+≡
```
    subroutine cputime_2 (u)
      integer, intent(in) :: u
      type(timer_t) :: timer

      write (u, "(A)")  "* Test output: cputime_2"
      write (u, "(A)")  "*   Purpose: check timer"
      write (u, "(A)")

      write (u, "(A)") "* Undefined timer"
      write (u, *)

      call timer%write (u)

      write (u, *)
      write (u, "(A)") "* Start timer"
      write (u, *)

      call timer%start ()
      call timer%write (u)

      write (u, *)
      write (u, "(A)") "* Stop timer (injecting fake timings)"
      write (u, *)

      call timer%stop ()
      timer%t1 = 2
      timer%t2 = 5
      call timer%evaluate ()
      call timer%write (u)

      write (u, *)
      write (u, "(A)") "* Restart timer"
      write (u, *)

      call timer%restart ()
```

```
        call timer%write (u)

        write (u, *)
        write (u, "(A)") "* Stop timer again (injecting fake timing)"
        write (u, *)

        call timer%stop ()
        timer%t2 = 10
        call timer%evaluate ()
        call timer%write (u)

        write (u, *)
        write (u, "(A)")  "* Test output end: cputime_2"

     end subroutine cputime_2
```

2.7 Hashtables

Hash tables, like lists, are not part of Fortran and must be defined on a per-case basis. In this section we define a module that contains a hash function.

Furthermore, for reference there is a complete framework of hashtable type definitions and access functions. This code is to be replicated where hash tables are used, mutatis mutandis.

⟨hashes.f90⟩≡
  ⟨*File header*⟩

```
  module hashes

    use kinds, only: i8, i32 !NODEP!
    use bytes
```

  ⟨*Standard module head*⟩

  ⟨*Hashes: public*⟩

```
  contains
```

  ⟨*Hashes: procedures*⟩

```
  end module hashes
```

2.7.1 The hash function

This is the one-at-a-time hash function by Bob Jenkins (from Wikipedia), re-implemented in Fortran. The function works on an array of bytes (8-bit integers), as could be produced by, e.g., the `transfer` function, and returns a single 32-bit integer. For determining the position in a hashtable, one can pick the lower bits of the result as appropriate to the hashtable size (which should be a power of 2). Note that we are working on signed integers, so the interpreta-

tion of values differs from the C version. This should not matter in practice, however.

⟨*Hashes: public*⟩≡

```
public :: hash
```

⟨*Hashes: procedures*⟩≡

```
function hash (key) result (hashval)
  integer(i32) :: hashval
  integer(i8), dimension(:), intent(in) :: key
  type(word32_t) :: w
  integer :: i
  w = 0_i32
  do i = 1, size (key)
     w = w + key(i)
     w = w + ishft (w, 10)
     w = ieor (w, ishft (w, -6))
  end do
  w = w + ishft (w, 3)
  w = ieor (w, ishft (w, -11))
  w = w + ishft (w, 15)
  hashval = w
end function hash
```

2.7.2   The hash table

We define a generic hashtable type (that depends on the `hash_data_t` type) together with associated methods.

This is a template:

⟨*Hashtables: types*⟩≡

```
type :: hash_data_t
   integer :: i
end type hash_data_t
```

Associated methods:

⟨*Hashtables: procedures*⟩≡

```
subroutine hash_data_final (data)
  type(hash_data_t), intent(inout) :: data
end subroutine hash_data_final

subroutine hash_data_write (data, unit)
  type(hash_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, *) data%i
end subroutine hash_data_write
```

Each hash entry stores the unmasked hash value, the key, and points to actual data if present. Note that this could be an allocatable scalar in principle, but making it a pointer avoids deep copy when expanding the hashtable.

⟨*Hashtables: types*⟩+≡

```
type :: hash_entry_t
   integer(i32) :: hashval = 0
   integer(i8), dimension(:), allocatable :: key
   type(hash_data_t), pointer :: data => null ()
end type hash_entry_t
```

The hashtable object holds the actual table, the number of filled entries and the number of entries after which the size should be doubled. The mask is equal to the table size minus one and thus coincides with the upper bound of the table index, which starts at zero.

⟨*Hashtables: types*⟩+≡

```
type :: hashtable_t
   integer :: n_entries = 0
   real :: fill_ratio = 0
   integer :: n_entries_max = 0
   integer(i32) :: mask = 0
   type(hash_entry_t), dimension(:), allocatable :: entry
end type hashtable_t
```

Initializer: The size has to be a power of two, the fill ratio is a real (machine-default!) number between 0 and 1.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_init (hashtable, size, fill_ratio)
  type(hashtable_t), intent(out) :: hashtable
  integer, intent(in) :: size
  real, intent(in) :: fill_ratio
  hashtable%fill_ratio = fill_ratio
  hashtable%n_entries_max = size * fill_ratio
  hashtable%mask = size - 1
  allocate (hashtable%entry (0:hashtable%mask))
end subroutine hashtable_init
```

Finalizer: This calls a `hash_data_final` subroutine which must exist.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_final (hashtable)
  type(hashtable_t), intent(inout) :: hashtable
  integer :: i
  do i = 0, hashtable%mask
     if (associated (hashtable%entry(i)%data)) then
        call hash_data_final (hashtable%entry(i)%data)
        deallocate (hashtable%entry(i)%data)
     end if
  end do
  deallocate (hashtable%entry)
end subroutine hashtable_final
```

Output. Here, we refer to a `hash_data_write` subroutine.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_write (hashtable, unit)
  type(hashtable_t), intent(in) :: hashtable
  integer, intent(in), optional :: unit
```

65

```
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      do i = 0, hashtable%mask
         if (associated (hashtable%entry(i)%data)) then
            write (u, *) i, "(hash =", hashtable%entry(i)%hashval, ")", &
                 hashtable%entry(i)%key
            call hash_data_write (hashtable%entry(i)%data, unit)
         end if
      end do
   end subroutine hashtable_write
```

2.7.3   Hashtable insertion

Insert a single entry with the hash value as trial place. If the table is filled, first expand it.

⟨*Hashtables: procedures*⟩+≡
```
   subroutine hashtable_insert (hashtable, key, data)
     type(hashtable_t), intent(inout) :: hashtable
     integer(i8), dimension(:), intent(in) :: key
     type(hash_data_t), intent(in), target :: data
     integer(i32) :: h
     if (hashtable%n_entries >= hashtable%n_entries_max) &
          call hashtable_expand (hashtable)
     h = hash (key)
     call hashtable_insert_rec (hashtable, h, h, key, data)
   end subroutine hashtable_insert
```

We need this auxiliary routine for doubling the size of the hashtable. We rely on the fact that default assignment copies the data pointer, not the data themselves. The temporary array must not be finalized; it is deallocated automatically together with its allocatable components.

⟨*Hashtables: procedures*⟩+≡
```
   subroutine hashtable_expand (hashtable)
     type(hashtable_t), intent(inout) :: hashtable
     type(hash_entry_t), dimension(:), allocatable :: table_tmp
     integer :: i, s
     allocate (table_tmp (0:hashtable%mask))
     table_tmp = hashtable%entry
     deallocate (hashtable%entry)
     s = 2 * size (table_tmp)
     hashtable%n_entries = 0
     hashtable%n_entries_max = s * hashtable%fill_ratio
     hashtable%mask = s - 1
     allocate (hashtable%entry (0:hashtable%mask))
     do i = 0, ubound (table_tmp, 1)
        if (associated (table_tmp(i)%data)) then
           call hashtable_insert_rec (hashtable, table_tmp(i)%hashval, &
                table_tmp(i)%hashval, table_tmp(i)%key, table_tmp(i)%data)
        end if
     end do
   end subroutine hashtable_expand
```

Insert a single entry at a trial place h, reduced to the table size. Collision resolution is done simply by choosing the next element, recursively until the place is empty. For bookkeeping, we preserve the original hash value. For a good hash function, there should be no clustering.

Note that if the new key exactly matches an existing key, nothing is done.

⟨*Hashtables: procedures*⟩+≡

```
recursive subroutine hashtable_insert_rec (hashtable, h, hashval, key, data)
  type(hashtable_t), intent(inout) :: hashtable
  integer(i32), intent(in) :: h, hashval
  integer(i8), dimension(:), intent(in) :: key
  type(hash_data_t), intent(in), target :: data
  integer(i32) :: i
  i = iand (h, hashtable%mask)
  if (associated (hashtable%entry(i)%data)) then
     if (size (hashtable%entry(i)%key) /= size (key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
     else if (any (hashtable%entry(i)%key /= key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
     end if
  else
     hashtable%entry(i)%hashval = hashval
     allocate (hashtable%entry(i)%key (size (key)))
     hashtable%entry(i)%key = key
     hashtable%entry(i)%data => data
     hashtable%n_entries = hashtable%n_entries + 1
  end if
end subroutine hashtable_insert_rec
```

2.7.4 Hashtable lookup

The lookup function has to parallel the insert function. If the place is filled, check if the key matches. Yes: return the pointer; no: increment the hash value and check again.

⟨*Hashtables: procedures*⟩+≡

```
function hashtable_lookup (hashtable, key) result (ptr)
  type(hash_data_t), pointer :: ptr
  type(hashtable_t), intent(in) :: hashtable
  integer(i8), dimension(:), intent(in) :: key
  ptr => hashtable_lookup_rec (hashtable, hash (key), key)
end function hashtable_lookup
```

⟨*Hashtables: procedures*⟩+≡

```
recursive function hashtable_lookup_rec (hashtable, h, key) result (ptr)
  type(hash_data_t), pointer :: ptr
  type(hashtable_t), intent(in) :: hashtable
  integer(i32), intent(in) :: h
  integer(i8), dimension(:), intent(in) :: key
  integer(i32) :: i
  i = iand (h, hashtable%mask)
  if (associated (hashtable%entry(i)%data)) then
     if (size (hashtable%entry(i)%key) == size (key)) then
```

```
            if (all (hashtable%entry(i)%key == key)) then
               ptr => hashtable%entry(i)%data
            else
               ptr => hashtable_lookup_rec (hashtable, h + 1, key)
            end if
         else
            ptr => hashtable_lookup_rec (hashtable, h + 1, key)
         end if
      else
         ptr => null ()
      end if
   end function hashtable_lookup_rec
```

⟨*Hashtables: public*⟩≡
```
      public :: hashtable_test
```

⟨*Hashtables: procedures*⟩+≡
```
      subroutine hashtable_test ()
        type(hash_data_t), pointer :: data
        type(hashtable_t) :: hashtable
        integer(i8) :: i
        call hashtable_init (hashtable, 16, 0.25)
        do i = 1, 10
           allocate (data)
           data%i = i*i
           call hashtable_insert (hashtable, [i, i+i], data)
        end do
        call hashtable_insert (hashtable, [2_i8, 4_i8], data)
        call hashtable_write (hashtable)
        data => hashtable_lookup (hashtable, [5_i8, 10_i8])
        if (associated (data)) then
           print *, "lookup:", data%i
        else
           print *, "lookup: --"
        end if
        data => hashtable_lookup (hashtable, [6_i8, 12_i8])
        if (associated (data)) then
           print *, "lookup:", data%i
        else
           print *, "lookup: --"
        end if
        data => hashtable_lookup (hashtable, [4_i8, 9_i8])
        if (associated (data)) then
           print *, "lookup:", data%i
        else
           print *, "lookup: --"
        end if
        call hashtable_final (hashtable)
      end subroutine hashtable_test
```

2.8 Message and signal handling

We are not so ambitious as to do proper exception handling in WHIZARD, but at least it may be useful to have a common interface for diagnostics: Results, messages, warnings, and such. As module variables we keep a buffer where the current message may be written to and a level indicator which tells which messages should be written on screen and which ones should be skipped. Alternatively, a string may be directly supplied to the message routine: this overrides the buffer, avoiding the necessety of formatted I/O in trivial cases.

⟨diagnostics.f90⟩≡

  ⟨*File header*⟩

```
  module diagnostics

    use iso_c_binding !NODEP!
    use system_dependencies !NODEP!
```
⟨*Use kinds*⟩
⟨*Use strings*⟩
```
    use limits, only: BUFFER_SIZE, MAX_ERRORS !NODEP!
    use file_utils !NODEP!
    use iso_fortran_env, only: &
       stdout => output_unit, stdin => input_unit !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*Diagnostics: public*⟩

  ⟨*Diagnostics: parameters*⟩

  ⟨*Diagnostics: types*⟩

  ⟨*Diagnostics: variables*⟩

  ⟨*Diagnostics: interfaces*⟩

```
  contains
```

  ⟨*Diagnostics: procedures*⟩

```
  end module diagnostics
```
Diagnostics levels:

⟨*Diagnostics: parameters*⟩≡
```
    integer, parameter :: &
        & TERMINATE=-2, BUG=-1, &
        & FATAL=1, ERROR=2, WARNING=3, MESSAGE=4, RESULT=5, DEBUG=6
```

⟨*Diagnostics: variables*⟩≡
```
    integer, save :: msg_level = RESULT
```

Mask fatal errors so that are treated as normal errors. Useful for interactive mode.

⟨*Diagnostics: public*⟩≡
```
    public :: mask_fatal_errors
```

⟨*Diagnostics: variables*⟩+≡
```
    logical, save :: mask_fatal_errors = .false.
```

How to handle bugs and unmasked fatal errors. Either execute a normal stop statement, or call the C `exit()` function, or try to cause a program crash by dereferencing a null pointer.

⟨*Diagnostics: parameters*⟩+≡
```
    integer, parameter :: TERM_STOP = 0, TERM_EXIT = 1, TERM_CRASH = 2
```

⟨*Diagnostics: variables*⟩+≡
```
    integer, save :: handle_fatal_errors = TERM_EXIT
```

Keep track of errors. This might be used for exception handling, later. The counter is incremented only for screen messages, to avoid double counting.

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_count
```

⟨*Diagnostics: variables*⟩+≡
```
    integer, dimension(TERMINATE:DEBUG), save :: msg_count = 0
```

Keep a list of all errors and warnings. Since we do not know the number of entries beforehand, we use a linked list.

⟨*Diagnostics: types*⟩≡
```
    type :: string_list
       character(len=BUFFER_SIZE) :: string
       type(string_list), pointer :: next
    end type string_list
    type :: string_list_pointer
       type(string_list), pointer :: first, last
    end type string_list_pointer
```

⟨*Diagnostics: variables*⟩+≡
```
    type(string_list_pointer), dimension(TERMINATE:WARNING), save :: &
         & msg_list = string_list_pointer (null(), null())
```

Add the current message buffer contents to the internal list.

⟨*Diagnostics: procedures*⟩≡
```
    subroutine msg_add (level)
      integer, intent(in) :: level
      type(string_list), pointer :: message
      select case (level)
      case (TERMINATE:WARNING)
         allocate (message)
         message%string = msg_buffer
         nullify (message%next)
         if (.not.associated (msg_list(level)%first)) &
              & msg_list(level)%first => message
         if (associated (msg_list(level)%last)) &
              & msg_list(level)%last%next => message
         msg_list(level)%last => message
         msg_count(level) = msg_count(level) + 1
      end select
    end subroutine msg_add
```

Initialization:

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_list_clear
```

⟨*Diagnostics: procedures*⟩+≡

```
    subroutine msg_list_clear
      integer :: level
      type(string_list), pointer :: message
      do level = TERMINATE, WARNING
         do while (associated (msg_list(level)%first))
            message => msg_list(level)%first
            msg_list(level)%first => message%next
            deallocate (message)
         end do
         nullify (msg_list(level)%last)
      end do
      msg_count = 0
    end subroutine msg_list_clear
```

Display the summary of errors and warnings (no need to count fatals...)

⟨*Diagnostics: public*⟩+≡

```
    public :: msg_summary
```

⟨*Diagnostics: procedures*⟩+≡

```
    subroutine msg_summary (unit)
      integer, intent(in), optional :: unit
      call expect_summary (unit)
  1   format (A,1x,I2,1x,A,I2,1x,A)
      if (msg_count(ERROR) > 0 .and. msg_count(WARNING) > 0) then
         write (msg_buffer, 1) "There were", &
               & msg_count(ERROR), "error(s) and  ", &
               & msg_count(WARNING), "warning(s)."
         call msg_message (unit=unit)
      else if (msg_count(ERROR) > 0) then
         write (msg_buffer, 1) "There were", &
               & msg_count(ERROR), "error(s) and no warnings."
         call msg_message (unit=unit)
      else if (msg_count(WARNING) > 0) then
         write (msg_buffer, 1) "There were no errors and  ", &
               & msg_count(WARNING), "warning(s)."
         call msg_message (unit=unit)
      end if
    end subroutine msg_summary
```

Print the list of all messages of a given level.

⟨*Diagnostics: public*⟩+≡

```
    public :: msg_listing
```

⟨*Diagnostics: procedures*⟩+≡

```
    subroutine msg_listing (level, unit, prefix)
      integer, intent(in) :: level
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: prefix
      type(string_list), pointer :: message
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (present (unit))  u = unit
      message => msg_list(level)%first
```

```
      do while (associated (message))
         if (present (prefix)) then
            write (u, "(A)") prefix // trim (message%string)
         else
            write (u, "(A)") trim (message%string)
         end if
         message => message%next
      end do
      flush (u)
   end subroutine msg_listing
```

There is a hard limit on the line length which we should export. This buffer size is used both by the message handler and by the lexer below.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: BUFFER_SIZE = 1000
```

The message buffer:

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_buffer
```

⟨*Diagnostics: variables*⟩+≡
```
    character(len=BUFFER_SIZE), save :: msg_buffer = " "
```

After a message is issued, the buffer should be cleared:

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine buffer_clear
      msg_buffer = " "
    end subroutine buffer_clear
```

The generic handler for messages. If the unit is omitted (or = 6), the message is written to standard output if the precedence if sufficiently high (as determined by the value of `msg_level`). If the string is omitted, the buffer is used. In any case, the buffer is cleared after printing. In accordance with FORTRAN custom, the first column in the output is left blank. For messages and warnings, an additional exclamation mark and a blank is prepended. Furthermore, each message is appended to the internal message list (without prepending anything).

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine message_print (level, string, str_arr, unit, logfile)
      integer, intent(in) :: level
      character(len=*), intent(in), optional :: string
      type(string_t), dimension(:), intent(in), optional :: str_arr
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: logfile
      type(string_t) :: prep_string, aux_string, head_footer
      integer :: lu, i
      logical :: severe, is_error
      severe = .false.
      head_footer  = "*************************************************************************
      aux_string = ""
      is_error = .false.
      select case (level)
      case (TERMINATE)
         prep_string = ""
```

```
case (BUG)
   prep_string  = "*** WHIZARD BUG: "
   aux_string   = "***              "
   severe = .true.
   is_error = .true.
case (FATAL)
   prep_string  = "*** FATAL ERROR: "
   aux_string   = "***              "
   severe = .true.
   is_error = .true.
case (ERROR)
   prep_string = "*** ERROR: "
   aux_string  = "***        "
   is_error = .true.
case (WARNING)
   prep_string = "Warning: "
case (MESSAGE, DEBUG)
   prep_string = "| "
case default
   prep_string = ""
end select
if (present(string))  msg_buffer = string
lu = log_unit
if (present(unit)) then
   if (unit /= stdout) then
      if (severe) write (unit, "(A)") char(head_footer)
      if (is_error) write (unit, "(A)") char(head_footer)
      write (unit, "(A,A)") char(prep_string), trim(msg_buffer)
      if (present (str_arr)) then
         do i = 1, size(str_arr)
            write (unit, "(A,A)") char(aux_string), char(trim(str_arr(i)))
         end do
      end if
      if (is_error) write (unit, "(A)") char(head_footer)
      if (severe) write (unit, "(A)") char(head_footer)
      flush (unit)
      lu = -1
   else if (level <= msg_level) then
      if (severe) print "(A)", char(head_footer)
      if (is_error) print "(A)", char(head_footer)
      print "(A,A)", char(prep_string), trim(msg_buffer)
      if (present (str_arr)) then
         do i = 1, size(str_arr)
            print "(A,A)", char(aux_string), char(trim(str_arr(i)))
         end do
      end if
      if (is_error) print "(A)", char(head_footer)
      if (severe) print "(A)", char(head_footer)
      flush (stdout)
      if (unit == log_unit)  lu = -1
   end if
else if (level <= msg_level) then
   if (severe) print "(A)", char(head_footer)
   if (is_error) print "(A)", char(head_footer)
```

```
        print "(A,A)", char(prep_string), trim(msg_buffer)
           if (present (str_arr)) then
              do i = 1, size(str_arr)
                 print "(A,A)", char(aux_string), char(trim(str_arr(i)))
              end do
           end if
        if (is_error) print "(A)", char(head_footer)
        if (severe) print "(A)", char(head_footer)
        flush (stdout)
     end if
     if (present (logfile)) then
        if (.not. logfile)  lu = -1
     end if
     if (logging .and. lu >= 0) then
        if (severe) write (lu, "(A)") char(head_footer)
        if (is_error) write (lu, "(A)") char(head_footer)
        write (lu, "(A,A)")  char(prep_string), trim(msg_buffer)
        if (present (str_arr)) then
           do i = 1, size(str_arr)
              write (lu, "(A,A)") char(aux_string), char(trim(str_arr(i)))
           end do
        end if
        if (is_error) write (lu, "(A)") char(head_footer)
        if (severe) write (lu, "(A)") char(head_footer)
        flush (lu)
     end if
     call msg_add (level)
     call buffer_clear
   end subroutine message_print
```

The specific handlers. In the case of fatal errors, bugs (failed assertions) and
normal termination execution is stopped. For non-fatal errors a message is
printed to standard output if no unit is given. Only if the number of MAX_ERRORS
errors is reached, we abort the program. There are no further actions in the
other cases, but this may change.

⟨*Diagnostics: public*⟩+≡
```
   public :: msg_terminate
   public :: msg_bug, msg_fatal, msg_error, msg_warning
   public :: msg_message, msg_result, msg_debug
```

⟨*Diagnostics: procedures*⟩+≡
```
   subroutine msg_terminate (string, unit, quit_code)
     integer, intent(in), optional :: unit
     character(len=*), intent(in), optional :: string
     integer, intent(in), optional :: quit_code
     integer(c_int) :: return_code
     call release_term_signals ()
     if (present (quit_code)) then
        return_code = quit_code
     else
        return_code = 0
     end if
     if (present (string)) &
         call message_print (MESSAGE, string, unit=unit)
```

```
      call msg_summary (unit)
      if (return_code == 0 .and. expect_failures /= 0) then
         return_code = 5
         call message_print (MESSAGE, &
              "WHIZARD run finished with 'expect' failure(s).", unit=unit)
      else if (return_code == 7) then
         call message_print (MESSAGE, &
              "WHIZARD run finished with failed self-test.", unit=unit)
      else
         call message_print (MESSAGE, "WHIZARD run finished.", unit=unit)
      end if
      call message_print (0, &
           "|==============================================================================|", unit=u
      call logfile_final ()
      call msg_list_clear ()
      if (return_code /= 0) then
         call exit (return_code)
      else
         !!! Should implement WHIZARD exit code (currently only via C)
         ! stop
         call exit (0)
      end if
   end subroutine msg_terminate

   subroutine msg_bug (string, arr, unit)
     integer, intent(in), optional :: unit
     character(len=*), intent(in), optional :: string
     type(string_t), dimension(:), intent(in), optional :: arr
     logical, pointer :: crash_ptr
     call message_print (BUG, string, arr, unit)
     call msg_summary (unit)
     select case (handle_fatal_errors)
     case (TERM_EXIT)
        call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
        call exit (-1_c_int)
     case (TERM_CRASH)
        print *, "*** Intentional crash ***"
        crash_ptr => null ()
        print *, crash_ptr
     end select
     stop "WHIZARD run aborted."
   end subroutine msg_bug

   recursive subroutine msg_fatal (string, arr, unit)
     integer, intent(in), optional :: unit
     character(len=*), intent(in), optional :: string
     type(string_t), dimension(:), intent(in), optional :: arr
     logical, pointer :: crash_ptr
     if (mask_fatal_errors) then
        call msg_error (string, arr, unit)
     else
        call message_print (FATAL, string, arr, unit)
        call msg_summary (unit)
        select case (handle_fatal_errors)
```

```
      case (TERM_EXIT)
         call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
         call exit (1_c_int)
      case (TERM_CRASH)
         print *, "*** Intentional crash ***"
         crash_ptr => null ()
         print *, crash_ptr
      end select
      stop "WHIZARD run aborted."
   end if
end subroutine msg_fatal

subroutine msg_error (string, arr, unit)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  call message_print (ERROR, string, arr, unit)
  if (msg_count(ERROR) >= MAX_ERRORS) then
     mask_fatal_errors = .false.
     call msg_fatal (" Too many errors encountered.")
  else if (.not.present(unit) .and. .not.mask_fatal_errors)  then
     call message_print (MESSAGE, "             (WHIZARD run continues)")
  end if
end subroutine msg_error

subroutine msg_warning (string, arr, unit)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  call message_print (WARNING, string, arr, unit)
end subroutine msg_warning

subroutine msg_message (string, unit, arr, logfile)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  logical, intent(in), optional :: logfile
  call message_print (MESSAGE, string, arr, unit, logfile)
end subroutine msg_message

subroutine msg_result (string, arr, unit, logfile)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  logical, intent(in), optional :: logfile
  call message_print (RESULT, string, arr, unit, logfile)
end subroutine msg_result

subroutine msg_debug (string, arr, unit)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  call message_print (DEBUG, string, arr, unit)
end subroutine msg_debug
```

Interface to the standard clib exit function

⟨*Diagnostics: interfaces*⟩≡
```
    interface
       subroutine exit (status) bind (C)
         use iso_c_binding !NODEP!
         integer(c_int), value :: status
       end subroutine exit
    end interface
```

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MAX_ERRORS = 10
```

Print the WHIZARD banner:

⟨*Diagnostics: public*⟩+≡
```
    public :: msg_banner
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine msg_banner (unit)
      integer, intent(in), optional :: unit
      integer :: n_proc
      call message_print (0, "|=======================================================================
      call message_print (0, "|
      call message_print (0, "|    WW              WW  WW  WW  WW  WWWWW      WW       WWWW    WWWW
      call message_print (0, "|     WW     WW     WW  WW  WW  WW    WW       WWWW    WW  WW   WW
      call message_print (0, "|      WW  WW WW  WW    WWWWWWW  WW    WW      WW  WW   WWWWW    WW
      call message_print (0, "|      WWWW    WWWW     WW  WW  WW    WW      WWWWWWWW  WW  WW   WW
      call message_print (0, "|       WW      WW     WW  WW  WW  WWWWW  WW      WW  WW    WW  WWWW
      call message_print (0, "|
      call message_print (0, "|
      call message_print (0, "|                                                 W
      call message_print (0, "|                                                sW
      call message_print (0, "|                                                WW
      call message_print (0, "|                                                sWW
      call message_print (0, "|                                                WWW
      call message_print (0, "|                                                wWWW
      call message_print (0, "|                                                wWWWW
      call message_print (0, "|                                               WW WW
      call message_print (0, "|                                               WW WW
      call message_print (0, "|                                              wWW WW
      call message_print (0, "|                                             wWW   WW
      call message_print (0, "|                                             WW   WW
      call message_print (0, "|                                             WW   WW
      call message_print (0, "|                                            WW    WW
      call message_print (0, "|                                            WW    WW
      call message_print (0, "|                                           WW    WW
      call message_print (0, "|                                           WW    WW
      call message_print (0, "|                   wwwwww                  WW    WW
      call message_print (0, "|                  WWWWWww                  WW    WW
      call message_print (0, "|                 WWWWWwwwww    WW          WW
      call message_print (0, "|                   wWWWwwwwwWW             WW
      call message_print (0, "|                  wWWWWWWWWWWwWWW          WW
      call message_print (0, "|                  wWWWWW       wW       WWWWWWW
      call message_print (0, "|                   WWWW        wW       WW  wWWWWWWWwww
```

77

```
         call message_print (0, "|                               WWWW                    wWWWWWWWwwww
         call message_print (0, "|                               WWWW                    WWWW      WWw
         call message_print (0, "|                               WWWWww                  WWWW
         call message_print (0, "|                                WWWwwww                 WWWW
         call message_print (0, "|                                 wWWWWwww        wWWWWW
         call message_print (0, "|                                       WwwwwwwwwWWW
         call message_print (0, "|
         call message_print (0, "|
         call message_print (0, "|
         call message_print (0, "|  by:    Wolfgang Kilian, Thorsten Ohl, Juergen Reuter
         call message_print (0, "|         with contributions from Christian Speckner
         call message_print (0, "|         Contact: <whizard@desy.de>
         call message_print (0, "|
         call message_print (0, "|  if you use WHIZARD please cite:
         call message_print (0, "|         W. Kilian, T. Ohl, J. Reuter,  Eur.Phys.J.C71 (2011) 1742
         call message_print (0, "|                                         [arXiv: 0708.4233 [hep-ph]]
         call message_print (0, "|         M. Moretti, T. Ohl, J. Reuter, arXiv: hep-ph/0102195
         call message_print (0, "|
         call message_print (0, "|================================================================
         call message_print (0, "|                                  WHIZARD " // WHIZARD_VERSION, unit=uni
         call message_print (0, "|================================================================
       end subroutine msg_banner
```

2.8.1 Logfile

All screen output should be duplicated in the logfile, unless requested otherwise.

⟨*Diagnostics: public*⟩+≡
```
    public :: logging
```

⟨*Diagnostics: variables*⟩+≡
```
    integer, save :: log_unit = -1
    logical, target, save :: logging = .false.
```

⟨*Diagnostics: public*⟩+≡
```
    public :: logfile_init
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine logfile_init (filename)
      type(string_t), intent(in) :: filename
      call msg_message ("Writing log to '" // char (filename) // "'")
      if (.not. logging)  call msg_message ("(Logging turned off.)")
      log_unit = free_unit ()
      open (file = char (filename), unit = log_unit, &
            action = "write", status = "replace")
    end subroutine logfile_init
```

⟨*Diagnostics: public*⟩+≡
```
    public :: logfile_final
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine logfile_final ()
      if (log_unit >= 0) then
         close (log_unit)
         log_unit = -1
```

78

```
      end if
   end subroutine logfile_final
```

This returns the valid logfile unit only if the default is write to screen, and if
`logfile` is not set false.

⟨*Diagnostics: public*⟩+≡
```
   public :: logfile_unit
```

⟨*Diagnostics: procedures*⟩+≡
```
   function logfile_unit (unit, logfile)
     integer :: logfile_unit
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: logfile
     if (logging) then
        if (present (unit)) then
           if (unit == stdout) then
              logfile_unit = log_unit
           else
              logfile_unit = -1
           end if
        else if (present (logfile)) then
           if (logfile) then
              logfile_unit = log_unit
           else
              logfile_unit = -1
           end if
        else
           logfile_unit = log_unit
        end if
     else
        logfile_unit = -1
     end if
   end function logfile_unit
```

2.8.2   Checking values

The `expect` function does not just check a value for correctness (actually, it
checks if a logical expression is true); it records its result here. If failures are
present when the program terminates, the exit code is nonzero.

⟨*Diagnostics: variables*⟩+≡
```
   integer, save :: expect_total = 0
   integer, save :: expect_failures = 0
```

⟨*Diagnostics: public*⟩+≡
```
   public :: expect_record
```

⟨*Diagnostics: procedures*⟩+≡
```
   subroutine expect_record (success)
     logical, intent(in) :: success
     expect_total = expect_total + 1
     if (.not. success)  expect_failures = expect_failures + 1
   end subroutine expect_record
```

⟨*Diagnostics: public*⟩+≡
```
    public :: expect_clear
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine expect_clear ()
      expect_total = 0
      expect_failures = 0
    end subroutine expect_clear
```

⟨*Diagnostics: public*⟩+≡
```
    public :: expect_summary
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine expect_summary (unit, force)
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: force
      logical :: force_output
      force_output = .false.;  if (present (force))  force_output = force
      if (expect_total /= 0 .or. force_output) then
          call msg_message ("Summary of value checks:", unit)
          write (msg_buffer, "(2x,A,1x,I0,1x,A,1x,A,1x,I0)") &
                "Failures:", expect_failures, "/", "Total:", expect_total
          call msg_message (unit=unit)
      end if
    end subroutine expect_summary
```

Helpers for converting integers into strings with minimal length.

⟨*Diagnostics: public*⟩+≡
```
    public :: int2string
    public :: int2char
    public :: int2fixed
```

⟨*Diagnostics: procedures*⟩+≡
```
    pure function int2fixed (i) result (c)
      integer, intent(in) :: i
      character(200) :: c
      c = ""
      write (c, *) i
      c = adjustl (c)
    end function int2fixed

    pure function int2string (i) result (s)
      integer, intent(in) :: i
      type (string_t) :: s
      s = trim (int2fixed (i))
    end function int2string

    pure function int2char (i) result (c)
      integer, intent(in) :: i
      character(len (trim (int2fixed (i)))) :: c
      c = int2fixed (i)
    end function int2char
```

Dito for reals.

⟨*Diagnostics: public*⟩+≡
```
    public :: real2string
    public :: real2char
    public :: real2fixed
```

⟨*Diagnostics: interfaces*⟩+≡
```
    interface real2string
       module procedure real2string_list, real2string_fmt
    end interface
    interface real2char
       module procedure real2char_list, real2char_fmt
    end interface
```

⟨*Diagnostics: procedures*⟩+≡
```
    pure function real2fixed (x, fmt) result (c)
      real(default), intent(in) :: x
      character(*), intent(in), optional :: fmt
      character(200) :: c
      c = ""
      write (c, *) x
      c = adjustl (c)
    end function real2fixed

    pure function real2fixed_fmt (x, fmt) result (c)
      real(default), intent(in) :: x
      character(*), intent(in) :: fmt
      character(200) :: c
      c = ""
      write (c, fmt)  x
      c = adjustl (c)
    end function real2fixed_fmt

    pure function real2string_list (x) result (s)
      real(default), intent(in) :: x
      type(string_t) :: s
      s = trim (real2fixed (x))
    end function real2string_list

    pure function real2string_fmt (x, fmt) result (s)
      real(default), intent(in) :: x
      character(*), intent(in) :: fmt
      type(string_t) :: s
      s = trim (real2fixed_fmt (x, fmt))
    end function real2string_fmt

    pure function real2char_list (x) result (c)
      real(default), intent(in) :: x
      character(len_trim (real2fixed (x))) :: c
      c = real2fixed (x)
    end function real2char_list

    pure function real2char_fmt (x, fmt) result (c)
      real(default), intent(in) :: x
      character(*), intent(in) :: fmt
```

```
      character(len_trim (real2fixed_fmt (x, fmt))) :: c
      c = real2fixed_fmt (x, fmt)
    end function real2char_fmt
```

Dito for complex values; we do not use the slightly ugly FORTRAN output form here but instead introduce our own. Ifort and Portland seem to have problems with this, therefore temporarily disable it.

⟨*CCC Diagnostics: public*⟩≡
```
    public :: cmplx2string
    public :: cmplx2char
```

⟨*CCC Diagnostics: procedures*⟩≡
```
    pure function cmplx2string (x) result (s)
      complex(default), intent(in) :: x
      type(string_t) :: s
      s = real2string (real (x, default))
      if (aimag (x) /= 0) s = s // " + " // real2string (aimag (x)) // " I"
    end function cmplx2string

    pure function cmplx2char (x) result (c)
      complex(default), intent(in) :: x
      character(len (char (cmplx2string (x)))) :: c
      c = char (cmplx2string (x))
    end function cmplx2char
```

2.8.3   Suppression of numerical noise

⟨*Diagnostics: public*⟩+≡
```
    public :: pacify
```

⟨*Diagnostics: interfaces*⟩+≡
```
    interface pacify
       module procedure pacify_real_default
       module procedure pacify_complex_default
    end interface pacify
```

⟨*Diagnostics: procedures*⟩+≡
```
    elemental subroutine pacify_real_default (x, tolerance)
      real(default), intent(inout) :: x
      real(default), intent(in) :: tolerance
      if (abs (x) < tolerance)  x = 0
    end subroutine pacify_real_default

    elemental subroutine pacify_complex_default (x, tolerance)
      complex(default), intent(inout) :: x
      real(default), intent(in) :: tolerance
      if (abs (real (x)) < tolerance)   &
           x = cmplx (0._default, aimag (x), kind=default)
      if (abs (aimag (x)) < tolerance)  &
           x = cmplx (real (x), 0._default, kind=default)
    end subroutine pacify_complex_default
```

2.8.4 Signal handling

Killing the program by external signals may leave the files written by it in an undefined state. This can be avoided by catching signals and deferring program termination. Instead of masking only critical sections, we choose to mask signals globally (done in the main program) and terminate the program at predefined checkpoints only. Checkpoints are after each command, within the sampling function (so the program can be terminated after each event), and after each iteration in the phase-space generation algorithm.

Signal handling is done via a C interface to the `sigaction` system call. When a signal is raised that has been masked by the handler, the corresponding variable is set to the value of the signal. The variables are visible from the C signal handler.

The signal SIGINT is for keyboard interrupt (ctrl-C), SIGTERM is for system interrupt, e.g., at shutdown. The SIGXCPU and SIGXFSZ signals may be issued by batch systems.

⟨*Diagnostics: public*⟩+≡
```
    public :: wo_sigint
    public :: wo_sigterm
    public :: wo_sigxcpu
    public :: wo_sigxfsz
```

⟨*Diagnostics: variables*⟩+≡
```
    integer(c_int), bind(C), volatile :: wo_sigint = 0
    integer(c_int), bind(C), volatile :: wo_sigterm = 0
    integer(c_int), bind(C), volatile :: wo_sigxcpu = 0
    integer(c_int), bind(C), volatile :: wo_sigxfsz = 0
```

Here are the interfaces to the C functions. The routine `mask_term_signals` forces termination signals to be delayed. `release_term_signals` restores normal behavior. However, the program can be terminated anytime by calling `terminate_now_if_signal` which inspects the signals and terminates the program if requested..

⟨*Diagnostics: public*⟩+≡
```
    public :: mask_term_signals
```

⟨*Diagnostics: procedures*⟩+≡
```
    subroutine mask_term_signals ()
      integer(c_int) :: status
      logical :: ok
      wo_sigint = 0
      ok = wo_mask_sigint () == 0
      if (.not. ok)  call msg_error ("Masking SIGINT failed")
      wo_sigterm = 0
      ok = wo_mask_sigterm () == 0
      if (.not. ok)  call msg_error ("Masking SIGTERM failed")
      wo_sigxcpu = 0
      ok = wo_mask_sigxcpu () == 0
      if (.not. ok)  call msg_error ("Masking SIGXCPU failed")
      wo_sigxfsz = 0
      ok = wo_mask_sigxfsz () == 0
      if (.not. ok)  call msg_error ("Masking SIGXFSZ failed")
```

```
      end subroutine mask_term_signals
```

⟨*Diagnostics: interfaces*⟩+≡
```
      interface
         integer(c_int) function wo_mask_sigint () bind(C)
           import
         end function wo_mask_sigint
      end interface
      interface
         integer(c_int) function wo_mask_sigterm () bind(C)
           import
         end function wo_mask_sigterm
      end interface
      interface
         integer(c_int) function wo_mask_sigxcpu () bind(C)
           import
         end function wo_mask_sigxcpu
      end interface
      interface
         integer(c_int) function wo_mask_sigxfsz () bind(C)
           import
         end function wo_mask_sigxfsz
      end interface
```

⟨*Diagnostics: public*⟩+≡
```
      public :: release_term_signals
```
⟨*Diagnostics: procedures*⟩+≡
```
      subroutine release_term_signals ()
        integer(c_int) :: status
        logical :: ok
        ok = wo_release_sigint () == 0
        if (.not. ok)  call msg_error ("Releasing SIGINT failed")
        ok = wo_release_sigterm () == 0
        if (.not. ok)  call msg_error ("Releasing SIGTERM failed")
        ok = wo_release_sigxcpu () == 0
        if (.not. ok)  call msg_error ("Releasing SIGXCPU failed")
        ok = wo_release_sigxfsz () == 0
        if (.not. ok)  call msg_error ("Releasing SIGXFSZ failed")
      end subroutine release_term_signals
```

⟨*Diagnostics: interfaces*⟩+≡
```
      interface
         integer(c_int) function wo_release_sigint () bind(C)
           import
         end function wo_release_sigint
      end interface
      interface
         integer(c_int) function wo_release_sigterm () bind(C)
           import
         end function wo_release_sigterm
      end interface
      interface
         integer(c_int) function wo_release_sigxcpu () bind(C)
```

```
        import
     end function wo_release_sigxcpu
  end interface
  interface
     integer(c_int) function wo_release_sigxfsz () bind(C)
        import
     end function wo_release_sigxfsz
  end interface
```

⟨*Diagnostics: public*⟩+≡
```
  public :: signal_is_pending
```

⟨*Diagnostics: procedures*⟩+≡
```
  function signal_is_pending () result (flag)
    logical :: flag
    flag = &
         wo_sigint /= 0 .or. &
         wo_sigterm /= 0 .or. &
         wo_sigxcpu /= 0 .or. &
         wo_sigxfsz /= 0
  end function signal_is_pending
```

⟨*Diagnostics: public*⟩+≡
```
  public :: terminate_now_if_signal
```

⟨*Diagnostics: procedures*⟩+≡
```
  subroutine terminate_now_if_signal ()
    if (wo_sigint /= 0) then
       call msg_terminate ("Signal SIGINT (keyboard interrupt) received.", &
          quit_code=int (wo_sigint))
    else if (wo_sigterm /= 0) then
       call msg_terminate ("Signal SIGTERM (termination signal) received.", &
          quit_code=int (wo_sigterm))
    else if (wo_sigxcpu /= 0) then
       call msg_terminate ("Signal SIGXCPU (CPU time limit exceeded) received.", &
          quit_code=int (wo_sigxcpu))
    else if (wo_sigxfsz /= 0) then
       call msg_terminate ("Signal SIGXFSZ (file size limit exceeded) received.", &
          quit_code=int (wo_sigxfsz))
    end if
  end subroutine terminate_now_if_signal
```

2.9   C wrapper for sigaction

This implements calls to `sigaction` and the appropriate signal handlers in C.

⟨`signal_interface.c`⟩≡
```
 /*
 ⟨File header⟩
 */
 #include <signal.h>
 #include <stdlib.h>
```

```c
extern int wo_sigint;
extern int wo_sigterm;
extern int wo_sigxcpu;
extern int wo_sigxfsz;

static void wo_handler_sigint (int sig) {
  wo_sigint = sig;
}

static void wo_handler_sigterm (int sig) {
  wo_sigterm = sig;
}

static void wo_handler_sigxcpu (int sig) {
  wo_sigxcpu = sig;
}

static void wo_handler_sigxfsz (int sig) {
  wo_sigxfsz = sig;
}

int wo_mask_sigint () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigint;
  return sigaction(SIGINT, &sa, NULL);
}

int wo_mask_sigterm () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigterm;
  return sigaction(SIGTERM, &sa, NULL);
}

int wo_mask_sigxcpu () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigxcpu;
  return sigaction(SIGXCPU, &sa, NULL);
}

int wo_mask_sigxfsz () {
  struct sigaction sa;
  sigset_t blocks;
```

```c
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigxfsz;
  return sigaction(SIGXFSZ, &sa, NULL);
}

int wo_release_sigint () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGINT, &sa, NULL);
}

int wo_release_sigterm () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGTERM, &sa, NULL);
}

int wo_release_sigxcpu () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGXCPU, &sa, NULL);
}

int wo_release_sigxfsz () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGXFSZ, &sa, NULL);
}
```

2.10   C wrapper for printf

The printf family of functions is implemented in C with an undefined number
of arguments. This is not supported by the bind(C) interface. We therefore
write wrappers for the versions of sprintf that we will actually use.

⟨sprintf_interface.c⟩≡
```c
  /*
  ⟨File header⟩
  */
  #include <stdio.h>

  int sprintf_none(char* str, const char* format) {
    return sprintf(str, format);
  }

  int sprintf_int(char* str, const char* format, int val) {
    return sprintf(str, format, val);
  }

  int sprintf_double(char* str, const char* format, double val) {
    return sprintf(str, format, val);
  }

  int sprintf_str(char* str, const char* format, const char* val) {
    return sprintf(str, format, val);
  }
```

⟨sprintf interfaces⟩≡
```fortran
      interface
        function sprintf_none (str, fmt) result (stat) bind(C)
          use iso_c_binding !NODEP!
          integer(c_int) :: stat
          character(c_char), dimension(*), intent(inout) :: str
          character(c_char), dimension(*), intent(in) :: fmt
        end function sprintf_none
      end interface

      interface
        function sprintf_int (str, fmt, val) result (stat) bind(C)
          use iso_c_binding !NODEP!
          integer(c_int) :: stat
          character(c_char), dimension(*), intent(inout) :: str
          character(c_char), dimension(*), intent(in) :: fmt
          integer(c_int), value :: val
        end function sprintf_int
      end interface

      interface
        function sprintf_double (str, fmt, val) result (stat) bind(C)
          use iso_c_binding !NODEP!
          integer(c_int) :: stat
          character(c_char), dimension(*), intent(inout) :: str
          character(c_char), dimension(*), intent(in) :: fmt
          real(c_double), value :: val
        end function sprintf_double
      end interface

      interface
        function sprintf_str(str, fmt, val) result (stat) bind(C)
```

```
      use iso_c_binding !NODEP!
      integer(c_int) :: stat
      character(c_char), dimension(*), intent(inout) :: str
      character(c_char), dimension(*), intent(in) :: fmt
      character(c_char), dimension(*), intent(in) :: val
    end function sprintf_str
  end interface
```

2.11   Interface for formatted I/O

For access to formatted printing (possibly input), we interface the C `printf` family of functions. There are two important issues here:

1. `printf` takes an arbitrary number of arguments, relying on the C stack. This is not interoperable. We interface it with C wrappers that output a single integer, real or string and restrict the allowed formats accordingly.

2. Restricting format strings is essential also for preventing format string attacks. Allowing arbitrary format string would create a real security hole in a Fortran program.

3. The string returned by `sprintf` must be allocated to the right size.

⟨`formats.f90`⟩≡
  ⟨*File header*⟩

```
  module formats

    use iso_c_binding !NODEP!
```
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use unit_tests
```

  ⟨*Standard module head*⟩

  ⟨*Formats: public*⟩

  ⟨*Formats: parameters*⟩

  ⟨*Formats: types*⟩

  ⟨*Formats: interfaces*⟩

```
  contains
```

  ⟨*Formats: procedures*⟩

  ⟨*Formats: tests*⟩

```
  end module formats
```

2.11.1 Parsing a C format string

The C format string contains characters and format conversion specifications. The latter are initiated by a `%` sign. If the next letter is also a `%`, a percent sign is printed and no conversion is done. Otherwise, a conversion is done and applied to the next argument in the argument list. First comes an optional flag (`#`, `0`, `-`, `+`, or space), an optional field width (decimal digits starting not with zero), an optional precision (period, then another decimal digit string), a length modifier (irrelevant for us, therefore not supported), and a conversion specifier: `d` or `i` for integer; `e`, `f`, `g` (also upper case) for double-precision real, `s` for a string.

We explicitly exclude all other conversion specifiers, and we check the specifiers against the actual arguments.

A type for passing arguments

This is a polymorphic type that can hold integer, real (double), and string arguments.

⟨*Formats: parameters*⟩≡
```
    integer, parameter :: ARGTYPE_NONE = 0
    integer, parameter :: ARGTYPE_LOG = 1
    integer, parameter :: ARGTYPE_INT = 2
    integer, parameter :: ARGTYPE_REAL = 3
    integer, parameter :: ARGTYPE_STR = 4
```

The integer and real entries are actually scalars, but we avoid relying on the allocatable-scalar feature and make them one-entry arrays. The character entry is a real array which is a copy of the string.

Logical values are mapped to strings (true or false), so this type parameter value is mostly unused.

⟨*Formats: public*⟩≡
```
    public :: sprintf_arg_t
```

⟨*Formats: types*⟩≡
```
    type :: sprintf_arg_t
      private
      integer :: type = ARGTYPE_NONE
      integer(c_int), dimension(:), allocatable :: ival
      real(c_double), dimension(:), allocatable :: rval
      character(c_char), dimension(:), allocatable :: sval
    end type sprintf_arg_t
```

⟨*Formats: public*⟩+≡
```
    public :: sprintf_arg_init
```

⟨*Formats: interfaces*⟩≡
```
    interface sprintf_arg_init
       module procedure sprintf_arg_init_log
       module procedure sprintf_arg_init_int
       module procedure sprintf_arg_init_real
       module procedure sprintf_arg_init_str
    end interface
```

⟨*Formats: procedures*⟩≡
```
    subroutine sprintf_arg_init_log (arg, lval)
      type(sprintf_arg_t), intent(out) :: arg
      logical, intent(in) :: lval
      arg%type = ARGTYPE_STR
      if (lval) then
         allocate (arg%sval (5))
         arg%sval = ['t', 'r', 'u', 'e', c_null_char]
      else
         allocate (arg%sval (6))
         arg%sval = ['f', 'a', 'l', 's', 'e', c_null_char]
      end if
    end subroutine sprintf_arg_init_log

    subroutine sprintf_arg_init_int (arg, ival)
      type(sprintf_arg_t), intent(out) :: arg
      integer, intent(in) :: ival
      arg%type = ARGTYPE_INT
      allocate (arg%ival (1))
      arg%ival = ival
    end subroutine sprintf_arg_init_int

    subroutine sprintf_arg_init_real (arg, rval)
      type(sprintf_arg_t), intent(out) :: arg
      real(default), intent(in) :: rval
      arg%type = ARGTYPE_REAL
      allocate (arg%rval (1))
      arg%rval = rval
    end subroutine sprintf_arg_init_real

    subroutine sprintf_arg_init_str (arg, sval)
      type(sprintf_arg_t), intent(out) :: arg
      type(string_t), intent(in) :: sval
      integer :: i
      arg%type = ARGTYPE_STR
      allocate (arg%sval (len (sval) + 1))
      do i = 1, len (sval)
         arg%sval(i) = extract (sval, i, i)
      end do
      arg%sval(len (sval) + 1) = c_null_char
    end subroutine sprintf_arg_init_str
```

⟨*Formats: procedures*⟩+≡
```
    subroutine sprintf_arg_write (arg, unit)
      type(sprintf_arg_t), intent(in) :: arg
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      select case (arg%type)
      case (ARGTYPE_NONE)
        write (u, *) "[none]"
      case (ARGTYPE_INT)
        write (u, "(1x,A,1x)", advance = "no")  "[int]"
        write (u, *)  arg%ival
```

```
      case (ARGTYPE_REAL)
        write (u, "(1x,A,1x)", advance = "no")  "[real]"
        write (u, *)  arg%rval
      case (ARGTYPE_STR)
        write (u, "(1x,A,1x,A)", advance = "no")  "[string]", '"'
        write (u, *)  arg%rval, '"'
      end select
    end subroutine sprintf_arg_write
```

Return an upper bound for the length of the printed version; in case of strings
the result is exact.

⟨*Formats: procedures*⟩+≡
```
    elemental function sprintf_arg_get_length (arg) result (length)
      integer :: length
      type(sprintf_arg_t), intent(in) :: arg
      select case (arg%type)
      case (ARGTYPE_INT)
         length = log10 (real (huge (arg%ival(1)))) + 2
      case (ARGTYPE_REAL)
         length = log10 (real (radix (arg%rval(1))) ** digits (arg%rval(1))) + 8
      case (ARGTYPE_STR)
         length = size (arg%sval)
      case default
         length = 0
      end select
    end function sprintf_arg_get_length
```

⟨*Formats: procedures*⟩+≡
```
    subroutine sprintf_arg_apply_sprintf (arg, fmt, result, actual_length)
      type(sprintf_arg_t), intent(in) :: arg
      character(c_char), dimension(:), intent(in) :: fmt
      character(c_char), dimension(:), intent(inout) :: result
      integer, intent(out) :: actual_length
      integer(c_int) :: ival
      real(c_double) :: rval
      select case (arg%type)
      case (ARGTYPE_NONE)
        actual_length = sprintf_none (result, fmt)
      case (ARGTYPE_INT)
        ival = arg%ival(1)
        actual_length = sprintf_int (result, fmt, ival)
      case (ARGTYPE_REAL)
        rval = arg%rval(1)
        actual_length = sprintf_double (result, fmt, rval)
      case (ARGTYPE_STR)
        actual_length = sprintf_str (result, fmt, arg%sval)
      case default
        call msg_bug ("sprintf_arg_apply_sprintf called with illegal type")
      end select
      if (actual_length < 0) then
         write (msg_buffer, *) "Format: '", fmt, "'"
         call msg_message ()
         write (msg_buffer, *) "Output: '", result, "'"
```

```
      call msg_message ()
      call msg_error ("I/O error in sprintf call")
      actual_length = 0
    end if
  end subroutine sprintf_arg_apply_sprintf
```

Container type for the output

There is a procedure which chops the format string into pieces that contain at most one conversion specifier. Pairing this with a `sprintf_arg` object, we get the actual input to the `sprintf` interface. The type below holds this input and can allocate the output string.

⟨*Formats: types*⟩+≡
```
  type :: sprintf_interface_t
    private
    character(c_char), dimension(:), allocatable :: input_fmt
    type(sprintf_arg_t) :: arg
    character(c_char), dimension(:), allocatable :: output_str
    integer :: output_str_len = 0
  end type sprintf_interface_t
```

⟨*Formats: procedures*⟩+≡
```
  subroutine sprintf_interface_init (intf, fmt, arg)
    type(sprintf_interface_t), intent(out) :: intf
    type(string_t), intent(in) :: fmt
    type(sprintf_arg_t), intent(in) :: arg
    integer :: fmt_len, i
    fmt_len = len (fmt)
    allocate (intf%input_fmt (fmt_len + 1))
    do i = 1, fmt_len
       intf%input_fmt(i) = extract (fmt, i, i)
    end do
    intf%input_fmt(fmt_len+1) = c_null_char
    intf%arg = arg
    allocate (intf%output_str (len (fmt) + sprintf_arg_get_length (arg) + 1))
  end subroutine sprintf_interface_init
```

⟨*Formats: procedures*⟩+≡
```
  subroutine sprintf_interface_write (intf, unit)
    type(sprintf_interface_t), intent(in) :: intf
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    write (u, *) "Format string = ", '"', intf%input_fmt, '"'
    write (u, "(1x,A,1x)", advance = "no") "Argument = "
    call sprintf_arg_write (intf%arg, unit)
    if (intf%output_str_len > 0) then
       write (u, *) "Result string = ", &
             '"', intf%output_str (1:intf%output_str_len), '"'
    end if
  end subroutine sprintf_interface_write
```

Return the output string:

⟨Formats: procedures⟩+≡

```
function sprintf_interface_get_result (intf) result (string)
  type(string_t) :: string
  type(sprintf_interface_t), intent(in) :: intf
  character(kind = c_char, len = max (intf%output_str_len, 0)) :: buffer
  integer :: i
  if (intf%output_str_len > 0) then
     do i = 1, intf%output_str_len
        buffer(i:i) = intf%output_str(i)
     end do
     string = buffer(1:intf%output_str_len)
  else
     string = ""
  end if
end function sprintf_interface_get_result
```

⟨Formats: procedures⟩+≡

```
subroutine sprintf_interface_apply_sprintf (intf)
  type(sprintf_interface_t), intent(inout) :: intf
  call sprintf_arg_apply_sprintf &
       (intf%arg, intf%input_fmt, intf%output_str, intf%output_str_len)
end subroutine sprintf_interface_apply_sprintf
```

Import the interfaces defined in the previous section:

⟨Formats: interfaces⟩+≡
  ⟨sprintf interfaces⟩

Scan the format string

Chop it into pieces that contain one conversion specifier each. The zero-th piece contains the part before the first specifier. Check the specifiers and allow only the subset that we support. Also check for an exact match between conversion specifiers and input arguments. The result is an allocated array of `sprintf_interface` object; each one contains a piece of the format string and the corresponding argument.

⟨Formats: procedures⟩+≡

```
subroutine chop_and_check_format_string (fmt, arg, intf)
  type(string_t), intent(in) :: fmt
  type(sprintf_arg_t), dimension(:), intent(in) :: arg
  type(sprintf_interface_t), dimension(:), intent(out), allocatable :: intf
  integer :: n_args, i
  type(string_t), dimension(:), allocatable :: split_fmt
  type(string_t) :: word, buffer, separator
  integer :: pos, length, l
  logical :: ok
  type(sprintf_arg_t) :: arg_null
  ok = .true.
  length = 0
  n_args = size (arg)
  allocate (split_fmt (0:n_args))
```

```
split_fmt = ""
buffer = fmt
SCAN_ARGS: do i = 1, n_args
   FIND_CONVERSION: do
       call split (buffer, word, "%", separator=separator)
       if (separator == "") then
          call msg_message ('"' // char (fmt) // '"')
          call msg_error ("C-formatting string: " &
              // "too few conversion specifiers in format string")
          ok = .false.;  exit SCAN_ARGS
       end if
       split_fmt(i-1) = split_fmt(i-1) // word
       if (extract (buffer, 1, 1) /= "%") then
          split_fmt(i) = "%"
          exit FIND_CONVERSION
       else
          split_fmt(i-1) = split_fmt(i-1) // "%"
       end if
   end do FIND_CONVERSION
   pos = verify (buffer, "#0-+ ")   ! Flag characters (zero or more)
   split_fmt(i) = split_fmt(i) // extract (buffer, 1, pos-1)
   buffer = remove (buffer, 1, pos-1)
   pos = verify (buffer, "123456890")  ! Field width
   word = extract (buffer, 1, pos-1)
   if (len (word) /= 0) then
     call read_int_from_string (word, len (word), l)
     length = length + l
   end if
   split_fmt(i) = split_fmt(i) // word
   buffer = remove (buffer, 1, pos-1)
   if (extract (buffer, 1, 1) == ".") then
      buffer = remove (buffer, 1, 1)
      pos = verify (buffer, "1234567890")   ! Precision
      split_fmt(i) = split_fmt(i) // "." // extract (buffer, 1, pos-1)
      buffer = remove (buffer, 1, pos-1)
   end if
   ! Length modifier would come here, but is not allowed
   select case (char (extract (buffer, 1, 1)))  ! conversion specifier
   case ("d", "i")
      if (arg(i)%type /= ARGTYPE_INT) then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
             // "argument type mismatch: integer value expected")
         ok = .false.;  exit SCAN_ARGS
      end if
   case ("e", "E", "f", "F", "g", "G")
      if (arg(i)%type /= ARGTYPE_REAL) then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
             // "argument type mismatch: real value expected")
         ok = .false.;  exit SCAN_ARGS
      end if
   case ("s")
      if (arg(i)%type /= ARGTYPE_STR) then
```

```
                call msg_message ('"' // char (fmt) // '"')
                call msg_error ("C-formatting string: " &
                     // "argument type mismatch: logical or string value expected")
                ok = .false.;  exit SCAN_ARGS
            end if
        case default
            call msg_message ('"' // char (fmt) // '"')
            call msg_error ("C-formatting string: " &
                 // "illegal or incomprehensible conversion specifier")
            ok = .false.;  exit SCAN_ARGS
        end select
        split_fmt(i) = split_fmt(i) // extract (buffer, 1, 1)
        buffer = remove (buffer, 1, 1)
    end do SCAN_ARGS
    if (ok) then
        FIND_EXTRA_CONVERSION: do
            call split (buffer, word, "%", separator=separator)
            split_fmt(n_args) = split_fmt(n_args) // word // separator
            if (separator == "")  exit FIND_EXTRA_CONVERSION
            if (extract (buffer, 1, 1) == "%") then
                split_fmt(n_args) = split_fmt(n_args) // "%"
                buffer = remove (buffer, 1, 1)
            else
                call msg_message ('"' // char (fmt) // '"')
                call msg_error ("C-formatting string: " &
                     // "too many conversion specifiers in format string")
                ok = .false.;  exit FIND_EXTRA_CONVERSION
            end if
        end do FIND_EXTRA_CONVERSION
        split_fmt(n_args) = split_fmt(n_args) // buffer
        allocate (intf (0:n_args))
        call sprintf_interface_init (intf(0), split_fmt(0), arg_null)
        do i = 1, n_args
            call sprintf_interface_init (intf(i), split_fmt(i), arg(i))
        end do
    else
        allocate (intf (0))
    end if
contains
  subroutine read_int_from_string (word, length, l)
    type(string_t), intent(in) :: word
    integer, intent(in) :: length
    integer, intent(out) :: l
    character(len=length) :: buffer
    buffer = word
    read (buffer, *) l
  end subroutine read_int_from_string
end subroutine chop_and_check_format_string
```

2.11.2 API

⟨*Formats: public*⟩+≡
```
    public :: sprintf
```

⟨*Formats: procedures*⟩+≡
```
    function sprintf (fmt, arg) result (string)
      type(string_t) :: string
      type(string_t), intent(in) :: fmt
      type(sprintf_arg_t), dimension(:), intent(in) :: arg
      type(sprintf_interface_t), dimension(:), allocatable :: intf
      integer :: i
      string = ""
      call chop_and_check_format_string (fmt, arg, intf)
      if (size (intf) > 0) then
         do i = 0, ubound (intf, 1)
            call sprintf_interface_apply_sprintf (intf(i))
            string = string // sprintf_interface_get_result (intf(i))
         end do
      end if
    end function sprintf
```

2.11.3 Test

⟨*Formats: public*⟩+≡
```
    public :: format_test
```

⟨*Formats: procedures*⟩+≡
```
    subroutine format_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Formats: execute tests*⟩
```
    end subroutine format_test
```

⟨*Formats: execute tests*⟩≡
```
    call test (format_1, "format_1", &
         "check formatting routines", &
         u, results)
```
⟨*Formats: tests*⟩≡
```
    subroutine format_1 (u)
      integer, intent(in) :: u
      write (u, "(A)")  "*** Test 1: a string ***"
      write (u, "(A)")
      call test_run (var_str("%s"), 1, [4], ['abcdefghij'], u)
      write (u, "(A)")  "*** Test 2: two integers ***"
      write (u, "(A)")
      call test_run (var_str("%d,%d"), 2, [2, 2], ['42', '13'], u)
      write (u, "(A)")  "*** Test 3: floating point number ***"
      write (u, "(A)")
      call test_run (var_str("%8.4f"), 1, [3], ['42567.12345'], u)
      write (u, "(A)")  "*** Test 4: general expression ***"
      call test_run (var_str("%g"), 1, [3], ['3.1415'], u)
      contains
        subroutine test_run (fmt, n_args, type, buffer, unit)
          type(string_t), intent(in) :: fmt
          integer, intent(in) :: n_args, unit
          logical :: lval
```

```
            integer :: ival
            real(default) :: rval
            integer :: i
            type(string_t) :: string
            type(sprintf_arg_t), dimension(:), allocatable :: arg
            integer, dimension(n_args), intent(in) :: type
            character(*), dimension(n_args), intent(in) :: buffer
            write (unit, "(A,A)")   "Format string :", char(fmt)
            write (unit, "(A,I1)")  "Number of args:", n_args
            allocate (arg (n_args))
            do i = 1, n_args
               write (unit, "(A,I1)")  "Argument (type ) = ", type(i)
               select case (type(i))
               case (ARGTYPE_LOG)
                  read (buffer(i), *)  lval
                  call sprintf_arg_init (arg(i), lval)
               case (ARGTYPE_INT)
                  read (buffer(i), *)  ival
                  call sprintf_arg_init (arg(i), ival)
               case (ARGTYPE_REAL)
                  read (buffer(i), *)  rval
                  call sprintf_arg_init (arg(i), rval)
               case (ARGTYPE_STR)
                  call sprintf_arg_init (arg(i), var_str (trim (buffer(i))))
               end select
            end do
            string = sprintf (fmt, arg)
            write (unit, "(A,A,A)")  "Result: '", char (string), "'"
            deallocate (arg)
          end subroutine test_run
      end subroutine format_1
```

2.12   Bytes and such

In a few instances we will need the notion of a byte (8-bit) and a word (32 bit),
even a 64-bit word. A block of 512 bit is also needed (for MD5).

   We rely on integers up to 64 bit being supported by the processor. The main
difference to standard integers is the interpretation as unsigned integers.

⟨bytes.f90⟩≡
  ⟨*File header*⟩

  module bytes

    use kinds, only: i8, i32, i64 !NODEP!
  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*Bytes: public*⟩

  ⟨*Bytes: types*⟩

                                   98
```

⟨*Bytes: parameters*⟩

⟨*Bytes: interfaces*⟩

```
contains
```

⟨*Bytes: procedures*⟩

```
end module bytes
```

2.12.1   8-bit words: bytes

This is essentially a wrapper around 8-bit integers. The wrapper emphasises their special interpretation as a sequence of bits. However, we interpret bytes as unsigned integers.

⟨*Bytes: public*⟩≡
```
    public :: byte_t
```

⟨*Bytes: types*⟩≡
```
    type :: byte_t
      private
      integer(i8) :: i
    end type byte_t
```

⟨*Bytes: public*⟩+≡
```
    public :: byte_zero
```

⟨*Bytes: parameters*⟩≡
```
    type(byte_t), parameter :: byte_zero = byte_t (0_i8)
```

Set a byte from 8-bit integer:

⟨*Bytes: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Bytes: interfaces*⟩≡
```
    interface assignment(=)
       module procedure set_byte_from_i8
    end interface
```

⟨*Bytes: procedures*⟩≡
```
    subroutine set_byte_from_i8 (b, i)
      type(byte_t), intent(out) :: b
      integer(i8), intent(in) :: i
      b%i = i
    end subroutine set_byte_from_i8
```

Write a byte in one of two formats: either as a hexadecimal number (two digits, default) or as a decimal number (one to three digits). The decimal version is nontrivial because bytes are unsigned integers. Optionally append a newline.

⟨*Bytes: public*⟩+≡
```
    public :: byte_write
```

⟨*Bytes: interfaces*⟩+≡
```
    interface byte_write
        module procedure byte_write_unit, byte_write_string
    end interface
```
⟨*Bytes: procedures*⟩+≡
```
    subroutine byte_write_unit (b, unit, decimal, newline)
      type(byte_t), intent(in), optional :: b
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: decimal, newline
      logical :: dc, nl
      type(word32_t) :: w
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      dc = .false.;  if (present (decimal))  dc = decimal
      nl = .false.;  if (present (newline))  nl = newline
      if (dc) then
         w = b
         write (u, '(I3)', advance='no')  w%i
      else
         write (u, '(z2.2)', advance='no')  b%i
      end if
      if (nl) write (u, *)
    end subroutine byte_write_unit
```

The string version is hex-only

⟨*Bytes: procedures*⟩+≡
```
    subroutine byte_write_string (b, s)
      type(byte_t), intent(in) :: b
      character(len=2), intent(inout) :: s
      write (s, '(z2.2)')  b%i
    end subroutine byte_write_string
```

2.12.2   32-bit words

This is not exactly a 32-bit integer. A word is to be filled with bytes, and it may be partially filled. The filling is done lowest-byte first, highest-byte last. We count the bits, so `fill` should be either 0, 8, 16, 24, or 32. In printing words, we correspondingly distinguish between printing zeros and printing blanks.

⟨*Bytes: public*⟩+≡
```
    public :: word32_t
```
⟨*Bytes: types*⟩+≡
```
    type :: word32_t
       private
       integer(i32) :: i
       integer :: fill = 0
    end type word32_t
```

Assignment: the word is filled by inserting a 32-bit integer

⟨*Bytes: interfaces*⟩+≡
```
    interface assignment(=)
```

```
    module procedure word32_set_from_i32
    module procedure word32_set_from_byte
  end interface
```

⟨*Bytes: procedures*⟩+≡
```
  subroutine word32_set_from_i32 (w, i)
    type(word32_t), intent(out) :: w
    integer(i32), intent(in) :: i
    w%i = i
    w%fill = 32
  end subroutine word32_set_from_i32
```

Reverse assignment to a 32-bit integer. We do not check the fill status.

⟨*Bytes: interfaces*⟩+≡
```
  interface assignment(=)
    module procedure i32_from_word32
  end interface
```

⟨*Bytes: procedures*⟩+≡
```
  subroutine i32_from_word32 (i, w)
    integer(i32), intent(out) :: i
    type(word32_t), intent(in) :: w
    i = w%i
  end subroutine i32_from_word32
```

Filling with a 8-bit integer is slightly tricky, because in this interpretation integers are unsigned.

⟨*Bytes: procedures*⟩+≡
```
  subroutine word32_set_from_byte (w, b)
    type(word32_t), intent(out) :: w
    type(byte_t), intent(in) :: b
    if (b%i >= 0_i8) then
        w%i = b%i
    else
        w%i = 2_i32*(huge(0_i8)+1_i32) + b%i
    end if
    w%fill = 32
  end subroutine word32_set_from_byte
```

Check the fill status

⟨*Bytes: public*⟩+≡
```
  public :: word32_empty, word32_filled, word32_fill
```

⟨*Bytes: procedures*⟩+≡
```
  function word32_empty (w)
    type(word32_t), intent(in) :: w
    logical :: word32_empty
    word32_empty = (w%fill == 0)
  end function word32_empty

  function word32_filled (w)
    type(word32_t), intent(in) :: w
    logical :: word32_filled
    word32_filled = (w%fill == 32)
```

```
      end function word32_filled

   function word32_fill (w)
      type(word32_t), intent(in) :: w
      integer :: word32_fill
      word32_fill = w%fill
   end function word32_fill
```

Partial assignment: append a byte to a partially filled word. (Note: no assignment if the word is filled, so check this before if necessary.)

⟨*Bytes: public*⟩+≡
```
   public :: word32_append_byte
```

⟨*Bytes: procedures*⟩+≡
```
   subroutine word32_append_byte (w, b)
      type(word32_t), intent(inout) :: w
      type(byte_t), intent(in) :: b
      type(word32_t) :: w1
      if (.not. word32_filled (w)) then
         w1 = b
         call mvbits (w1%i, 0, 8, w%i, w%fill)
         w%fill = w%fill + 8
      end if
   end subroutine word32_append_byte
```

Extract a byte from a word. The argument i is the position, which may be 0, 1, 2, or 3.

For the final assignment, we set the highest bit separately. Otherwise, we might trigger an overflow condition for a compiler with strict checking turned on.

⟨*Bytes: public*⟩+≡
```
   public :: byte_from_word32
```

⟨*Bytes: procedures*⟩+≡
```
   function byte_from_word32 (w, i) result (b)
      type(word32_t), intent(in) :: w
      integer, intent(in) :: i
      type(byte_t) :: b
      integer(i32) :: j
      j = 0
      if (i >= 0 .and. i*8 < w%fill) then
         call mvbits (w%i, i*8, 8, j, 0)
      end if
      b%i = int (ibclr (j, 7), kind=i8)
      if (btest (j, 7))  b%i = ibset (b%i, 7)
   end function byte_from_word32
```

Write a word to file or STDOUT. We understand words as unsigned integers, therefore we cannot use the built-in routine unchanged. However, we can make use of the existence of 64-bit integers and their output routine.

In hexadecimal format, the default version prints eight hex characters, highest-first. The bytes version prints four bytes (two-hex characters), lowest first, with

spaces in-between. The decimal bytes version is analogous. In the `bytes` version, missing bytes are printed as whitespace.

⟨*Bytes: public*⟩+≡
```
    public :: word32_write
```

⟨*Bytes: interfaces*⟩+≡
```
    interface word32_write
       module procedure word32_write_unit
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    subroutine word32_write_unit (w, unit, bytes, decimal, newline)
      type(word32_t), intent(in) :: w
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: bytes, decimal, newline
      logical :: dc, by, nl
      type(word64_t) :: ww
      integer :: i, u
      u = output_unit (unit);  if (u < 0)  return
      by = .false.;  if (present (bytes))  by = bytes
      dc = .false.;  if (present (decimal)) dc = decimal
      nl = .false.;  if (present (newline)) nl = newline
      if (by) then
         do i = 0, 3
            if (i>0)  write (u, '(1x)', advance='no')
            if (8*i < w%fill) then
               call byte_write (byte_from_word32 (w, i), unit, decimal=decimal)
            else if (dc) then
               write (u, '(3x)', advance='no')
            else
               write (u, '(2x)', advance='no')
            end if
         end do
      else if (dc) then
         ww = w
         write (u, '(I10)', advance='no') ww%i
      else
         select case (w%fill)
         case ( 0)
         case ( 8);  write (6, '(1x,z8.2)', advance='no') ibits (w%i, 0, 8)
         case (16);  write (6, '(1x,z8.4)', advance='no') ibits (w%i, 0,16)
         case (24);  write (6, '(1x,z8.6)', advance='no') ibits (w%i, 0,24)
         case (32);  write (6, '(1x,z8.8)', advance='no') ibits (w%i, 0,32)
         end select
      end if
      if (nl) write (u, *)
    end subroutine word32_write_unit
```

2.12.3 Operations on 32-bit words

Define the usual logical operations, as well as addition (mod 2^{32}). We assume that all operands are completely filled.

⟨*Bytes: public*⟩+≡

```
      public :: not, ior, ieor, iand, ishft, ishftc
```
⟨*Bytes: interfaces*⟩+≡
```
      interface not
         module procedure word_not
      end interface
      interface ior
         module procedure word_or
      end interface
      interface ieor
         module procedure word_eor
      end interface
      interface iand
         module procedure word_and
      end interface
      interface ishft
         module procedure word_shft
      end interface
      interface ishftc
         module procedure word_shftc
      end interface
```
⟨*Bytes: procedures*⟩+≡
```
      function word_not (w1) result (w2)
        type(word32_t), intent(in) :: w1
        type(word32_t) :: w2
        w2 = not (w1%i)
      end function word_not

      function word_or (w1, w2) result (w3)
        type(word32_t), intent(in) :: w1, w2
        type(word32_t) :: w3
        w3 = ior (w1%i, w2%i)
      end function word_or

      function word_eor (w1, w2) result (w3)
        type(word32_t), intent(in) :: w1, w2
        type(word32_t) :: w3
        w3 = ieor (w1%i, w2%i)
      end function word_eor

      function word_and (w1, w2) result (w3)
        type(word32_t), intent(in) :: w1, w2
        type(word32_t) :: w3
        w3 = iand (w1%i, w2%i)
      end function word_and

      function word_shft (w1, s) result (w2)
        type(word32_t), intent(in) :: w1
        integer, intent(in) :: s
        type(word32_t) :: w2
        w2 = ishft (w1%i, s)
      end function word_shft

      function word_shftc (w1, s) result (w2)
```

```
      type(word32_t), intent(in) :: w1
      integer, intent(in) :: s
      type(word32_t) :: w2
      w2 = ishftc (w1%i, s, 32)
    end function word_shftc
```

Addition is defined mod 2^{32}, i.e., without overflow checking. This means that
we have to work around a possible overflow check enforced by the compiler.

⟨*Bytes: public*⟩+≡
```
    public :: operator(+)
```

⟨*Bytes: interfaces*⟩+≡
```
    interface operator(+)
       module procedure word_add
       module procedure word_add_i8
       module procedure word_add_i32
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    function word_add (w1, w2) result (w3)
      type(word32_t), intent(in) :: w1, w2
      type(word32_t) :: w3
      integer(i64) :: j
      j = int (ibclr (w1%i, 31), i64) + int (ibclr (w2%i, 31), i64)
      w3 = int (ibclr (j, 31), kind=i32)
      if (btest (j, 31)) then
         if (btest (w1%i, 31) .eqv. btest (w2%i, 31))  w3 = ibset (w3%i, 31)
      else
         if (btest (w1%i, 31) .neqv. btest (w2%i, 31))  w3 = ibset (w3%i, 31)
      end if
    end function word_add

    function word_add_i8 (w1, i) result (w3)
      type(word32_t), intent(in) :: w1
      integer(i8), intent(in) :: i
      type(word32_t) :: w3
      integer(i64) :: j
      j = int (ibclr (w1%i, 31), i64) + int (ibclr (i, 7), i64)
      if (btest (i, 7))  j = j + 128
      w3 = int (ibclr (j, 31), kind=i32)
      if (btest (j, 31) .neqv. btest (w1%i, 31))  w3 = ibset (w3%i, 31)
    end function word_add_i8

    function word_add_i32 (w1, i) result (w3)
      type(word32_t), intent(in) :: w1
      integer(i32), intent(in) :: i
      type(word32_t) :: w3
      integer(i64) :: j
      j = int (ibclr (w1%i, 31), i64) + int (ibclr (i, 31), i64)
      w3 = int (ibclr (j, 31), kind=i32)
      if (btest (j, 31)) then
         if (btest (w1%i, 31) .eqv. btest (i, 31))  w3 = ibset (w3%i, 31)
      else
         if (btest (w1%i, 31) .neqv. btest (i, 31))  w3 = ibset (w3%i, 31)
```

```
        end if
    end function word_add_i32
```

2.12.4   64-bit words

These objects consist of two 32-bit words. They thus can hold integer numbers larger than 2^{32} (to be exact, 2^{31} since FORTRAN integers are signed). The order is low-word, high-word.

⟨*Bytes: public*⟩+≡
```
    public :: word64_t
```

⟨*Bytes: types*⟩+≡
```
    type :: word64_t
       private
       integer(i64) :: i
    end type word64_t
```

Set a 64 bit word:

⟨*Bytes: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure word64_set_from_i64
       module procedure word64_set_from_word32
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    subroutine word64_set_from_i64 (ww, i)
      type(word64_t), intent(out) :: ww
      integer(i64), intent(in) :: i
      ww%i = i
    end subroutine word64_set_from_i64
```

Filling with a 32-bit word:

⟨*Bytes: procedures*⟩+≡
```
    subroutine word64_set_from_word32 (ww, w)
      type(word64_t), intent(out) :: ww
      type(word32_t), intent(in) :: w
      if (w%i >= 0_i32) then
         ww%i = w%i
      else
         ww%i = 2_i64*(huge(0_i32)+1_i64) + w%i
      end if
    end subroutine word64_set_from_word32
```

Extract a byte from a word. The argument `i` is the position, which may be between 0 and 7.

For the final assignment, we set the highest bit separately. Otherwise, we might trigger an overflow condition for a compiler with strict checking turned on.

⟨*Bytes: public*⟩+≡
```
    public :: byte_from_word64, word32_from_word64
```

⟨*Bytes: procedures*⟩+≡
```
    function byte_from_word64 (ww, i) result (b)
      type(word64_t), intent(in) :: ww
      integer, intent(in) :: i
      type(byte_t) :: b
      integer(i64) :: j
      j = 0
      if (i >= 0 .and. i*8 < 64) then
          call mvbits (ww%i, i*8, 8, j, 0)
      end if
      b%i = int (ibclr (j, 7), kind=i8)
      if (btest (j, 7))  b%i = ibset (b%i, 7)
    end function byte_from_word64
```

Extract a 32-bit word from a 64-bit word. The position is either 0 or 1.

⟨*Bytes: procedures*⟩+≡
```
    function word32_from_word64 (ww, i) result (w)
      type(word64_t), intent(in) :: ww
      integer, intent(in) :: i
      type(word32_t) :: w
      integer(i64) :: j
      j = 0
      select case (i)
      case (0);  call mvbits (ww%i,  0, 32, j, 0)
      case (1);  call mvbits (ww%i, 32, 32, j, 0)
      end select
      w = int (ibclr (j, 31), kind=i32)
      if (btest (j, 31))  w = ibset (w%i, 31)
    end function word32_from_word64
```

Print a 64-bit word. Decimal version works up to 2^{63}. The `words` version uses the 'word32' printout, separated by two spaces. The low-word is printed first. The `bytes` version also uses the 'word32' printout. This implies that the lowest byte is first. The default version prints a hexadecimal number without spaces, highest byte first.

⟨*Bytes: public*⟩+≡
```
    public :: word64_write
```

⟨*Bytes: interfaces*⟩+≡
```
    interface word64_write
       module procedure word64_write_unit
    end interface
```

⟨*Bytes: procedures*⟩+≡
```
    subroutine word64_write_unit (ww, unit, words, bytes, decimal, newline)
      type(word64_t), intent(in) :: ww
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: words, bytes, decimal, newline
      logical :: wo, by, dc, nl
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      wo = .false.;  if (present (words))    wo = words
      by = .false.;  if (present (bytes))    by = bytes
      dc = .false.;  if (present (decimal))  dc = decimal
```

```
      nl = .false.;  if (present (newline))  nl = newline
      if (wo .or. by) then
         call word32_write_unit (word32_from_word64 (ww, 0), unit, by, dc)
         write (u, '(2x)', advance='no')
         call word32_write_unit (word32_from_word64 (ww, 1), unit, by, dc)
      else if (dc) then
         write (u, '(I19)', advance='no') ww%i
      else
         write (u, '(Z16)', advance='no') ww%i
      end if
      if (nl) write (u, *)
   end subroutine word64_write_unit
```

2.13   MD5 Checksums

Implementing MD5 checksums allows us to check input/file integrity on the basis of a well-known standard. The building blocks have been introduced in the bytes module.

⟨md5.f90⟩≡
  ⟨*File header*⟩

```
  module md5

    use kinds, only: i8, i32, i64 !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use bytes
    use limits, only: BUFFER_SIZE, LF, EOR, EOF !NODEP!
    use unit_tests
```

  ⟨*Standard module head*⟩

  ⟨*MD5: public*⟩

  ⟨*MD5: types*⟩

  ⟨*MD5: variables*⟩

  ⟨*MD5: interfaces*⟩

```
  contains
```

  ⟨*MD5: procedures*⟩

  ⟨*MD5: tests*⟩

```
  end module md5
```

2.13.1 Blocks

A block is a sequence of 16 words (64 bytes or 512 bits). We anticipate that blocks will be linked, so include a pointer to the next block. There is a fill status (word counter), as there is one for each word. The fill status is equal to the number of bytes that are in, so it may be between 0 and 64.

⟨*MD5: types*⟩≡

```
type :: block_t
   private
   type(word32_t), dimension(0:15) :: w
   type(block_t), pointer :: next => null ()
   integer :: fill = 0
end type block_t
```

Check if a block is completely filled or empty:

⟨*MD5: procedures*⟩≡

```
function block_is_empty (b)
  type(block_t), intent(in) :: b
  logical :: block_is_empty
  block_is_empty = (b%fill == 0 .and. word32_empty (b%w(0)))
end function block_is_empty

function block_is_filled (b)
  type(block_t), intent(in) :: b
  logical :: block_is_filled
  block_is_filled = (b%fill == 64)
end function block_is_filled
```

Append a single byte to a block. Works only if the block is not yet filled.

⟨*MD5: procedures*⟩+≡

```
subroutine block_append_byte (bl, by)
  type(block_t), intent(inout) :: bl
  type(byte_t), intent(in) :: by
  if (.not. block_is_filled (bl)) then
     call word32_append_byte (bl%w(bl%fill/4), by)
     bl%fill = bl%fill + 1
  end if
end subroutine block_append_byte
```

The printing routine allows for printing as sequences of words or bytes, decimal or hex.

⟨*MD5: interfaces*⟩≡

```
interface block_write
   module procedure block_write_unit
end interface
```

⟨*MD5: procedures*⟩+≡

```
subroutine block_write_unit (b, unit, bytes, decimal)
  type(block_t), intent(in) :: b
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal
  logical :: by, dc
  integer :: i, u
```

```
      u = output_unit (unit);  if (u < 0)  return
      by = .false.;  if (present (bytes))    by = bytes
      dc = .false.;  if (present (decimal))  dc = decimal
      do i = 0, b%fill/4 - 1
         call newline_or_blank (u, i, by, dc)
         call word32_write (b%w(i), unit, bytes, decimal)
      end do
      if (.not. block_is_filled (b)) then
         i = b%fill/4
         if (.not. word32_empty (b%w(i))) then
            call newline_or_blank (u, i, by, dc)
            call word32_write (b%w(i), unit, bytes, decimal)
         end if
      end if
      write (u, *)
   contains
      subroutine newline_or_blank (u, i, bytes, decimal)
         integer, intent(in) :: u, i
         logical, intent(in) :: bytes, decimal
         if (decimal) then
            select case (i)
            case (0)
            case (2,4,6,8,10,12,14);  write (u, *)
            case default
               write (u, '(2x)', advance='no')
            end select
         else if (bytes) then
            select case (i)
            case (0)
            case (4,8,12);  write (u, *)
            case default
               write (u, '(2x)', advance='no')
            end select
         else
            if (i == 8)  write (u, *)
         end if
      end subroutine newline_or_blank
   end subroutine block_write_unit
```

2.13.2   Messages

A message (within this module) is a linked list of blocks.

⟨*MD5: types*⟩+≡

```
   type :: message_t
      private
      type(block_t), pointer :: first => null ()
      type(block_t), pointer :: last => null ()
      integer :: n_blocks = 0
   end type message_t
```

Clear the message list

⟨*MD5: procedures*⟩+≡

```
    subroutine message_clear (m)
      type(message_t), intent(inout) :: m
      type(block_t), pointer :: b
      nullify (m%last)
      do
         b => m%first
         if (.not.(associated (b))) exit
         m%first => b%next
         deallocate (b)
      end do
      m%n_blocks = 0
    end subroutine message_clear
```

Append an empty block to the message list

⟨MD5: procedures⟩+≡
```
    subroutine message_append_new_block (m)
      type(message_t), intent(inout) :: m
      if (associated (m%last)) then
         allocate (m%last%next)
         m%last => m%last%next
         m%n_blocks = m%n_blocks + 1
      else
         allocate (m%first)
         m%last => m%first
         m%n_blocks = 1
      end if
    end subroutine message_append_new_block
```

Initialize: clear and allocate the first (empty) block.

⟨MD5: procedures⟩+≡
```
    subroutine message_init (m)
      type(message_t), intent(inout) :: m
      call message_clear (m)
      call message_append_new_block (m)
    end subroutine message_init
```

Append a single byte to a message. If necessary, allocate a new block. If the
message is empty, initialize it.

⟨MD5: procedures⟩+≡
```
    subroutine message_append_byte (m, b)
      type(message_t), intent(inout) :: m
      type(byte_t), intent(in) :: b
      if (.not. associated (m%last)) then
         call message_init (m)
      else if (block_is_filled (m%last)) then
         call message_append_new_block (m)
      end if
      call block_append_byte (m%last, b)
    end subroutine message_append_byte
```

Append zero bytes until the current block is filled up to the required position.
If we are already beyond that, append a new block and fill that one.

⟨*MD5: procedures*⟩+≡
```
    subroutine message_pad_zero (m, i)
      type(message_t), intent(inout) :: m
      integer, intent(in) :: i
      type(block_t), pointer :: b
      integer :: j
      if (associated (m%last)) then
         b => m%last
         if (b%fill > i) then
            do j = b%fill + 1, 64 + i
               call message_append_byte (m, byte_zero)
            end do
         else
            do j = b%fill + 1, i
               call message_append_byte (m, byte_zero)
            end do
         end if
      end if
    end subroutine message_pad_zero
```

This returns the number of bits within a message. We need a 64-bit word for the result since it may be more than 2^{31}. This is also required by the MD5 standard.

⟨*MD5: procedures*⟩+≡
```
    function message_bits (m) result (length)
      type(message_t), intent(in) :: m
      type(word64_t) :: length
      type(block_t), pointer :: b
      integer(i64) :: n_blocks_filled, n_bytes_extra
      if (m%n_blocks > 0) then
         b => m%last
         if (block_is_filled (b)) then
            n_blocks_filled = m%n_blocks
            n_bytes_extra = 0
         else
            n_blocks_filled = m%n_blocks - 1
            n_bytes_extra = b%fill
         end if
         length = n_blocks_filled * 512 + n_bytes_extra * 8
      else
         length = 0_i64
      end if
    end function message_bits
```

2.13.3 Message I/O

Append the contents of a string to a message. We first cast the character string into a 8-bit integer array and the append this byte by byte.

⟨*MD5: procedures*⟩+≡
```
    subroutine message_append_string (m, s)
      type(message_t), intent(inout) :: m
      character(len=*), intent(in) :: s
```

```
      integer(i64) :: i, n_bytes
      integer(i8), dimension(:), allocatable :: buffer
      integer(i8), dimension(1) :: mold
      type(byte_t) :: b
      n_bytes = size (transfer (s, mold))
      allocate (buffer (n_bytes))
      buffer = transfer (s, mold)
      do i = 1, size (buffer)
         b = buffer(i)
         call message_append_byte (m, b)
      end do
      deallocate (buffer)
   end subroutine message_append_string
```

Append the contents of a 32-bit integer to a message. We first cast the 32-bit integer into a 8-bit integer array and the append this byte by byte.

⟨*MD5: procedures*⟩+≡
```
   subroutine message_append_i32 (m, x)
     type(message_t), intent(inout) :: m
     integer(i32), intent(in) :: x
     integer(i8), dimension(4) :: buffer
     type(byte_t) :: b
     integer :: i
     buffer = transfer (x, buffer, size(buffer))
     do i = 1, size (buffer)
        b = buffer(i)
        call message_append_byte (m, b)
     end do
   end subroutine message_append_i32
```

Append one line from file to a message. Include the newline character.

⟨*MD5: procedures*⟩+≡
```
   subroutine message_append_from_unit (m, u, iostat)
     type(message_t), intent(inout) :: m
     integer, intent(in) :: u
     integer, intent(out) :: iostat
     character(len=BUFFER_SIZE) :: buffer
     read (u, *, iostat=iostat) buffer
     call message_append_string (m, trim (buffer))
     call message_append_string (m, LF)
   end subroutine message_append_from_unit
```

Fill a message from file. (Each line counts as a string.)

⟨*MD5: procedures*⟩+≡
```
   subroutine message_read_from_file (m, f)
     type(message_t), intent(inout) :: m
     character(len=*), intent(in) :: f
     integer :: u, iostat
     u = free_unit ()
     open (file=f, unit=u, action='read')
     do
        call message_append_from_unit (m, u, iostat=iostat)
```

```
      if (iostat < 0) exit
    end do
    close (u)
  end subroutine message_read_from_file
```

Write a message. After each block, insert an empty line.

⟨*MD5: interfaces*⟩+≡
```
    interface message_write
       module procedure message_write_unit
    end interface
```

⟨*MD5: procedures*⟩+≡
```
    subroutine message_write_unit (m, unit, bytes, decimal)
      type(message_t), intent(in) :: m
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: bytes, decimal
      type(block_t), pointer :: b
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      b => m%first
      if (associated (b)) then
         do
            call block_write_unit (b, unit, bytes, decimal)
            b => b%next
            if (.not. associated (b))  exit
            write (u, *)
         end do
      end if
    end subroutine message_write_unit
```

2.13.4 Auxiliary functions

These four functions on three words are defined in the MD5 standard:

⟨*MD5: procedures*⟩+≡
```
    function ff (x, y, z)
      type(word32_t), intent(in) :: x, y, z
      type(word32_t) :: ff
      ff = ior (iand (x, y), iand (not (x), z))
    end function ff

    function fg (x, y, z)
      type(word32_t), intent(in) :: x, y, z
      type(word32_t) :: fg
      fg = ior (iand (x, z), iand (y, not (z)))
    end function fg

    function fh (x, y, z)
      type(word32_t), intent(in) :: x, y, z
      type(word32_t) :: fh
      fh = ieor (ieor (x, y), z)
    end function fh
```

```
function fi (x, y, z)
  type(word32_t), intent(in) :: x, y, z
  type(word32_t) :: fi
  fi = ieor (y, ior (x, not (z)))
end function fi
```

2.13.5   Auxiliary stuff

This defines and initializes the table of transformation constants:

⟨*MD5: variables*⟩≡
```
type(word32_t), dimension(64), save :: t
logical, save :: table_initialized = .false.
```

⟨*MD5: procedures*⟩+≡
```
subroutine table_init
  type(word64_t) :: ww
  integer :: i
  if (.not.table_initialized) then
     do i = 1, 64
        ww = int (4294967296d0 * abs (sin (i * 1d0)), kind=i64)
        t(i) = word32_from_word64 (ww, 0)
     end do
     table_initialized = .true.
  end if
end subroutine table_init
```

This encodes the message digest (4 words) into a 32-character string.

⟨*MD5: procedures*⟩+≡
```
function digest_string (aa) result (s)
  type(word32_t), dimension (0:3), intent(in) :: aa
  character(len=32) :: s
  integer :: i, j
  do i = 0, 3
     do j = 0, 3
        call byte_write (byte_from_word32 (aa(i), j), s(i*8+j*2+1:i*8+j*2+2))
     end do
  end do
end function digest_string
```

2.13.6   MD5 algorithm

Pad the message with a byte x80 and then pad zeros up to a full block minus two
words; in these words, insert the message length (before padding) as a 64-bit
word, low-word first.

⟨*MD5: procedures*⟩+≡
```
subroutine message_pad (m)
  type(message_t), intent(inout) :: m
  type(word64_t) :: length
  integer(i8), parameter :: ipad = -128  ! z'80'
  type(byte_t) :: b
```

```
      integer :: i
      length = message_bits (m)
      b = ipad
      call message_append_byte (m, b)
      call message_pad_zero (m, 56)
      do i = 0, 7
          call message_append_byte (m, byte_from_word64 (length, i))
      end do
  end subroutine message_pad
```

Apply a series of transformations onto a state a,b,c,d, where the transform function uses each word of the message together with the predefined words. Finally, encode the state as a 32-character string.

⟨*MD5: procedures*⟩+≡
```
  subroutine message_digest (m, s)
    type(message_t), intent(in) :: m
    character(len=32), intent(out) :: s
    integer(i32), parameter :: ia =  1732584193  ! z'67452301'
    integer(i32), parameter :: ib =  -271733879  ! z'efcdab89'
    integer(i32), parameter :: ic = -1732584194  ! z'98badcfe'
    integer(i32), parameter :: id =   271733878  ! z'10325476'
    type(word32_t) :: a, b, c, d
    type(word32_t) :: aa, bb, cc, dd
    type(word32_t), dimension(0:15) :: x
    type(block_t), pointer :: bl
    call table_init
    a = ia;  b = ib;  c = ic;  d = id
    bl => m%first
    do
        if (.not.associated (bl)) exit
        x = bl%w
        aa = a;  bb = b;  cc = c;  dd = d
        call transform (ff, a, b, c, d,  0,  7,  1)
        call transform (ff, d, a, b, c,  1, 12,  2)
        call transform (ff, c, d, a, b,  2, 17,  3)
        call transform (ff, b, c, d, a,  3, 22,  4)
        call transform (ff, a, b, c, d,  4,  7,  5)
        call transform (ff, d, a, b, c,  5, 12,  6)
        call transform (ff, c, d, a, b,  6, 17,  7)
        call transform (ff, b, c, d, a,  7, 22,  8)
        call transform (ff, a, b, c, d,  8,  7,  9)
        call transform (ff, d, a, b, c,  9, 12, 10)
        call transform (ff, c, d, a, b, 10, 17, 11)
        call transform (ff, b, c, d, a, 11, 22, 12)
        call transform (ff, a, b, c, d, 12,  7, 13)
        call transform (ff, d, a, b, c, 13, 12, 14)
        call transform (ff, c, d, a, b, 14, 17, 15)
        call transform (ff, b, c, d, a, 15, 22, 16)
        call transform (fg, a, b, c, d,  1,  5, 17)
        call transform (fg, d, a, b, c,  6,  9, 18)
        call transform (fg, c, d, a, b, 11, 14, 19)
        call transform (fg, b, c, d, a,  0, 20, 20)
        call transform (fg, a, b, c, d,  5,  5, 21)
```

```
      call transform (fg, d, a, b, c, 10,  9, 22)
      call transform (fg, c, d, a, b, 15, 14, 23)
      call transform (fg, b, c, d, a,  4, 20, 24)
      call transform (fg, a, b, c, d,  9,  5, 25)
      call transform (fg, d, a, b, c, 14,  9, 26)
      call transform (fg, c, d, a, b,  3, 14, 27)
      call transform (fg, b, c, d, a,  8, 20, 28)
      call transform (fg, a, b, c, d, 13,  5, 29)
      call transform (fg, d, a, b, c,  2,  9, 30)
      call transform (fg, c, d, a, b,  7, 14, 31)
      call transform (fg, b, c, d, a, 12, 20, 32)
      call transform (fh, a, b, c, d,  5,  4, 33)
      call transform (fh, d, a, b, c,  8, 11, 34)
      call transform (fh, c, d, a, b, 11, 16, 35)
      call transform (fh, b, c, d, a, 14, 23, 36)
      call transform (fh, a, b, c, d,  1,  4, 37)
      call transform (fh, d, a, b, c,  4, 11, 38)
      call transform (fh, c, d, a, b,  7, 16, 39)
      call transform (fh, b, c, d, a, 10, 23, 40)
      call transform (fh, a, b, c, d, 13,  4, 41)
      call transform (fh, d, a, b, c,  0, 11, 42)
      call transform (fh, c, d, a, b,  3, 16, 43)
      call transform (fh, b, c, d, a,  6, 23, 44)
      call transform (fh, a, b, c, d,  9,  4, 45)
      call transform (fh, d, a, b, c, 12, 11, 46)
      call transform (fh, c, d, a, b, 15, 16, 47)
      call transform (fh, b, c, d, a,  2, 23, 48)
      call transform (fi, a, b, c, d,  0,  6, 49)
      call transform (fi, d, a, b, c,  7, 10, 50)
      call transform (fi, c, d, a, b, 14, 15, 51)
      call transform (fi, b, c, d, a,  5, 21, 52)
      call transform (fi, a, b, c, d, 12,  6, 53)
      call transform (fi, d, a, b, c,  3, 10, 54)
      call transform (fi, c, d, a, b, 10, 15, 55)
      call transform (fi, b, c, d, a,  1, 21, 56)
      call transform (fi, a, b, c, d,  8,  6, 57)
      call transform (fi, d, a, b, c, 15, 10, 58)
      call transform (fi, c, d, a, b,  6, 15, 59)
      call transform (fi, b, c, d, a, 13, 21, 60)
      call transform (fi, a, b, c, d,  4,  6, 61)
      call transform (fi, d, a, b, c, 11, 10, 62)
      call transform (fi, c, d, a, b,  2, 15, 63)
      call transform (fi, b, c, d, a,  9, 21, 64)
      a = a + aa
      b = b + bb
      c = c + cc
      d = d + dd
      bl => bl%next
    end do
  s = digest_string ([a, b, c, d])
contains
⟨MD5: Internal subroutine transform⟩
end subroutine message_digest
```

And this is the actual transformation that depends on one of the previous functions, four words, and three integers. The implicit arguments are `x`, the word from the message to digest, and `t`, the entry in the predefined table.

⟨*MD5: Internal subroutine transform*⟩≡

```
subroutine transform (f, a, b, c, d, k, s, i)
  interface
     function f (x, y, z)
       import word32_t
       type(word32_t), intent(in) :: x, y, z
       type(word32_t) :: f
     end function f
  end interface
  type(word32_t), intent(inout) :: a
  type(word32_t), intent(in) :: b, c, d
  integer, intent(in) :: k, s, i
  a = b + ishftc (a + f(b, c, d) + x(k) + t(i), s)
end subroutine transform
```

2.13.7   User interface

⟨*MD5: public*⟩≡

```
public :: md5sum
```

⟨*MD5: interfaces*⟩+≡

```
interface md5sum
   module procedure md5sum_from_string
   module procedure md5sum_from_unit
end interface
```

This function computes the MD5 sum of the input string and returns it as a 32-character string

⟨*MD5: procedures*⟩+≡

```
function md5sum_from_string (s) result (digest)
  character(len=*), intent(in) :: s
  character(len=32) :: digest
  type(message_t) :: m
  call message_append_string (m, s)
  call message_pad (m)
  call message_digest (m, digest)
  call message_clear (m)
end function md5sum_from_string
```

This funct. reads from unit u (an unformmated sequence of integers) and computes the MD5 sum.

⟨*MD5: procedures*⟩+≡

```
function md5sum_from_unit (u) result (digest)
  integer, intent(in) :: u
  character(len=32) :: digest
  type(message_t) :: m
  character :: char
  integer :: iostat
  READ_CHARS: do
```

```
                 read (u, "(A)", advance="no", iostat=iostat)  char
                 select case (iostat)
                 case (0)
                    call message_append_string (m, char)
                 case (EOR)
                    call message_append_string (m, LF)
                 case (EOF)
                    exit READ_CHARS
                 case default
                    call msg_fatal &
                         ("Computing MD5 sum: I/O error while reading from scratch file")
                 end select
              end do READ_CHARS
              call message_pad (m)
              call message_digest (m, digest)
              call message_clear (m)
            end function md5sum_from_unit
```

2.13.8   Unit tests

⟨*MD5: public*⟩+≡
```
    public :: md5_test
```

⟨*MD5: procedures*⟩+≡
```
    subroutine md5_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨MD5: execute tests⟩
    end subroutine md5_test
```

This function checks the implementation by computing the checksum of certain
strings and comparing them with the known values.

⟨*MD5: execute tests*⟩≡
```
    call test (md5_1, "md5_1", &
         "check MD5 sums", &
         u, results)
```

⟨*MD5: tests*⟩≡
```
    subroutine md5_1 (u)
      integer, intent(in) :: u
      character(32) :: s
      integer, parameter :: n = 7
      integer :: i
      character(80), dimension(n) :: teststring
      data teststring(1) /""/
      data teststring(2) /"a"/
      data teststring(3) /"abc"/
      data teststring(4) /"message digest"/
      data teststring(5) /"abcdefghijklmnopqrstuvwxyz"/
      data teststring(6) /"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"/
      data teststring(7) /"123456789012345678901234567890123456789012345678901234567890123
      character(32), dimension(n) :: result
      data result(1) /"D41D8CD98F00B204E9800998ECF8427E"/
```

```
      data result(2) /"0CC175B9C0F1B6A831C399E269772661"/
      data result(3) /"900150983CD24FB0D6963F7D28E17F72"/
      data result(4) /"F96B697D7CB7938D525A2F31AAF161D0"/
      data result(5) /"C3FCD3D76192E4007DFB496CCA67E13B"/
      data result(6) /"D174AB98D277D9F5A5611C2C9F419D9F"/
      data result(7) /"57EDF4A22BE3C955AC49DA2E2107B67A"/

      write (u, "(A)")  "* Test output: MD5"
      write (u, "(A)")  "*   Purpose: test MD5 sums"
      write (u, "(A)")

      do i = 1, n
         write (u, "(A)") "MD5 test string = " // '"'// &
               trim (teststring(i)) // '"'
         s = md5sum (trim (teststring(i)))
         write (u, "(A)") "MD5 check sum   = " // trim (s)
         write (u, "(A)") "Ref check sum   = " // result(i)
         if (s == result(i)) then
            call msg_message ("=> ok", u)
         else
            call msg_message ("=> MD5 sum self-test failed", u)
         end if
      end do
      call msg_message ("========================================================================
    end subroutine md5_1
```

2.14   Permutations

⟨permutations.f90⟩≡
  ⟨*File header*⟩

```
  module permutations

     use kinds, only: TC !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*Permutations: public*⟩

  ⟨*Permutations: types*⟩

  ⟨*Permutations: interfaces*⟩

```
  contains
```

  ⟨*Permutations: procedures*⟩

```
  end module permutations
```

2.14.1 Permutations

A permutation is an array of integers. Each integer between one and `size` should occur exactly once.

⟨*Permutations: public*⟩≡
```
    public :: permutation_t
```

⟨*Permutations: types*⟩≡
```
    type :: permutation_t
      private
      integer, dimension(:), allocatable :: p
    end type permutation_t
```

Initialize with the identity permutation.

⟨*Permutations: public*⟩+≡
```
    public :: permutation_init
    public :: permutation_final
```

⟨*Permutations: procedures*⟩≡
```
    elemental subroutine permutation_init (p, size)
      type(permutation_t), intent(inout) :: p
      integer, intent(in) :: size
      integer :: i
      allocate (p%p (size))
      forall (i = 1:size)
         p%p(i) = i
      end forall
    end subroutine permutation_init

    elemental subroutine permutation_final (p)
      type(permutation_t), intent(inout) :: p
      deallocate (p%p)
    end subroutine permutation_final
```

I/O:

⟨*Permutations: public*⟩+≡
```
    public :: permutation_write
```

⟨*Permutations: procedures*⟩+≡
```
    subroutine permutation_write (p, u)
      type(permutation_t), intent (in) :: p
      integer, intent(in) :: u
      integer :: i
      do i = 1, size (p%p)
         if (size (p%p) < 10) then
            write (u,"(1x,I1)", advance="no") p%p(i)
         else
            write (u,"(1x,I3)", advance="no") p%p(i)
         end if
      end do
      write (u, *)
    end subroutine permutation_write
```

Administration:

⟨*Permutations: public*⟩+≡
```
    public :: permutation_size
```

⟨*Permutations: procedures*⟩+≡
```
    elemental function permutation_size (perm) result (s)
      type(permutation_t), intent(in) :: perm
      integer :: s
      s = size (perm%p)
    end function permutation_size
```

Extract an entry in a permutation.

⟨*Permutations: public*⟩+≡
```
    public :: permute
```

⟨*Permutations: procedures*⟩+≡
```
    elemental function permute (i, p) result (j)
      integer, intent(in) :: i
      type(permutation_t), intent(in) :: p
      integer :: j
      if (i > 0 .and. i <= size (p%p)) then
         j = p%p(i)
      else
         j = 0
      end if
    end function permute
```

Check whether a permutation is valid: Each integer in the range occurs exactly once.

⟨*Permutations: public*⟩+≡
```
    public :: permutation_ok
```

⟨*Permutations: procedures*⟩+≡
```
    elemental function permutation_ok (perm) result (ok)
      type(permutation_t), intent(in) :: perm
      logical :: ok
      integer :: i
      logical, dimension(:), allocatable :: set
      ok = .true.
      allocate (set (size (perm%p)))
      set = .false.
      do i = 1, size (perm%p)
         ok = (perm%p(i) > 0 .and. perm%p(i) <= size (perm%p))
         if (.not.ok) return
         set(perm%p(i)) = .true.
      end do
      ok = all (set)
    end function permutation_ok
```

Find the permutation that transforms the second array into the first one. We assume that this is possible and unique and all bounds are set correctly.

This cannot be elemental.

⟨*Permutations: public*⟩+≡
```
    public :: permutation_find
```

⟨*Permutations: procedures*⟩+≡

```fortran
subroutine permutation_find (perm, a1, a2)
  type(permutation_t), intent(inout) :: perm
  integer, dimension(:), intent(in) :: a1, a2
  integer :: i, j
  if (allocated (perm%p))  deallocate (perm%p)
  allocate (perm%p (size (a1)))
  do i = 1, size (a1)
     do j = 1, size (a2)
        if (a1(i) == a2(j)) then
           perm%p(i) = j
           exit
        end if
        perm%p(i) = 0
     end do
  end do
end subroutine permutation_find
```

Find all permutations that transform an array of integers into itself. The resulting permutation list is allocated with the correct length and filled.

The first step is to count the number of different entries in `code`. Next, we scan `code` again and assign a mask to each different entry, true for all identical entries. Finally, we recursively permute the identity for each possible mask.

The permutation is done as follows: A list of all permutations of the initial one with respect to the current mask is generated, then the permutations are generated in turn for each permutation in this list with the next mask. The result is always stored back into the main list, starting from the end of the current list.

⟨*Permutations: public*⟩+≡

```fortran
public :: permutation_array_make
```

⟨*Permutations: procedures*⟩+≡

```fortran
subroutine permutation_array_make (pa, code)
  type(permutation_t), dimension(:), allocatable, intent(out) :: pa
  integer, dimension(:), intent(in) :: code
  logical, dimension(size(code)) :: mask
  logical, dimension(:,:), allocatable :: imask
  integer, dimension(:), allocatable :: n_i
  type(permutation_t) :: p_init
  type(permutation_t), dimension(:), allocatable :: p_tmp
  integer :: psize, i, j, k, n_different, n, nn_k
  psize = size (code)
  mask = .true.
  n_different = 0
  do i=1, psize
     if (mask(i)) then
        n_different = n_different + 1
        mask = mask .and. (code /= code(i))
     end if
  end do
  allocate (imask(psize, n_different), n_i(n_different))
  mask = .true.
  k = 0
```

123

```
      do i=1, psize
         if (mask(i)) then
            k = k + 1
            imask(:,k) = (code == code(i))
            n_i(k) = factorial (count(imask(:,k)))
            mask = mask .and. (code /= code(i))
         end if
      end do
      n = product (n_i)
      allocate (pa (n))
      call permutation_init (p_init, psize)
      pa(1) = p_init
      nn_k = 1
      do k = 1, n_different
         allocate (p_tmp (n_i(k)))
         do i = nn_k, 1, -1
            call permutation_array_with_mask (p_tmp, imask(:,k), pa(i))
            do j = n_i(k), 1, -1
               pa((i-1)*n_i(k) + j) = p_tmp(j)
            end do
         end do
         deallocate (p_tmp)
         nn_k = nn_k * n_i(k)
      end do
      call permutation_final (p_init)
      deallocate (imask, n_i)
    end subroutine permutation_array_make
```

Make a list of permutations of the elements marked true in the `mask` array. The final permutation list must be allocated with the correct length ($n!$). The third argument is the initial permutation to start with, which must have the same length as the `mask` array (this is not checked).

⟨*Permutations: procedures*⟩+≡

```
    subroutine permutation_array_with_mask (pa, mask, p_init)
      type(permutation_t), dimension(:), intent(inout) :: pa
      logical, dimension(:), intent(in) :: mask
      type(permutation_t), intent(in) :: p_init
      integer :: plen
      integer :: i, ii, j, fac_i, k, x
      integer, dimension(:), allocatable :: index
      plen = size (pa)
      allocate (index(count(mask)))
      ii = 0
      do i = 1, size (mask)
         if (mask(i)) then
            ii = ii + 1
            index(ii) = i
         end if
      end do
      pa = p_init
      ii = 0
      fac_i = 1
      do i = 1, size (mask)
```

```
          if (mask(i)) then
              ii = ii + 1
              fac_i = fac_i * ii
              x = permute (i, p_init)
              do j = 1, plen
                  k = ii - mod (((j-1)*fac_i)/plen, ii)
                  call insert (pa(j), x, k, ii, index)
              end do
          end if
      end do
      deallocate (index)
    contains
      subroutine insert (p, x, k, n, index)
        type(permutation_t), intent(inout) :: p
        integer, intent(in) :: x, k, n
        integer, dimension(:), intent(in) :: index
        integer :: i
        do i = n, k+1, -1
            p%p(index(i)) = p%p(index(i-1))
        end do
        p%p(index(k)) = x
      end subroutine insert
    end subroutine permutation_array_with_mask
```

The factorial function is needed for pre-determining the number of permutations
that will be generated:

⟨*Permutations: procedures*⟩+≡
```
    function factorial (n) result (f)
      integer, intent(in) :: n
      integer :: f
      integer :: i
      f = 1
      do i=2, abs(n)
          f = f*i
      end do
    end function factorial
```

2.14.2   Operations on binary codes

Binary codes are needed for phase-space trees. Since the permutation function
uses permutations, and no other special type is involved, we put the functions
here.

This is needed for phase space trees: permute bits in a tree binary code. If
no permutation is given, leave as is. (We may want to access the permutation
directly here if this is efficiency-critical.)

⟨*Permutations: public*⟩+≡
```
    public :: tc_permute
```

⟨*Permutations: procedures*⟩+≡
```
    function tc_permute (k, perm, mask_in) result (pk)
      integer(TC), intent(in) :: k, mask_in
      type(permutation_t), intent(in) :: perm
```

```
      integer(TC) :: pk
      integer :: i
      pk = iand (k, mask_in)
      do i = 1, size (perm%p)
         if (btest(k,i-1))  pk = ibset (pk, perm%p(i)-1)
      end do
   end function tc_permute
```

This routine returns the number of set bits in the tree code value `k`. Hence, it is the number of externals connected to the current line. If `mask` is present, the complement of the tree code is also considered, and the smaller number is returned. This gives the true distance from the external states, taking into account the initial particles. The complement number is increased by one, since for a scattering diagram the vertex with the sum of all final-state codes is still one point apart from the initial particles.

⟨*Permutations: public*⟩+≡
```
    public :: tc_decay_level
```

⟨*Permutations: interfaces*⟩≡
```
    interface tc_decay_level
       module procedure decay_level_simple
       module procedure decay_level_complement
    end interface
```

⟨*Permutations: procedures*⟩+≡
```
    function decay_level_complement (k, mask) result (l)
      integer(TC), intent(in) :: k, mask
      integer :: l
      l = min (decay_level_simple (k), &
           &   decay_level_simple (ieor (k, mask)) + 1)
    end function decay_level_complement

    function decay_level_simple (k) result(l)
      integer(TC), intent(in) :: k
      integer :: l
      integer :: i
      l = 0
      do i=0, bit_size(k)-1
         if (btest(k,i)) l = l+1
      end do
    end function decay_level_simple
```

2.15 Sorting

This small module provides functions for sorting integer or real arrays.

⟨`sorting.f90`⟩≡
  ⟨*File header*⟩

  module sorting

  ⟨*Use kinds*⟩

```
    use diagnostics !NODEP!
    use unit_tests
```

⟨*Standard module head*⟩

⟨*Sorting: public*⟩

⟨*Sorting: interfaces*⟩

```
contains
```

⟨*Sorting: procedures*⟩

⟨*Sorting: tests*⟩

```
end module sorting
```

2.15.1 Implementation

The `sort` function returns, for a given integer or real array, the array sorted by increasing value. The current implementation is *mergesort*, which has $O(n \ln n)$ behavior in all cases, and is stable for elements of equal value.

The `sort_abs` variant sorts by increasing absolute value, where for identical absolute value, the positive number comes first.

⟨*Sorting: public*⟩≡
```
    public :: sort
    public :: sort_abs
```

⟨*Sorting: interfaces*⟩≡
```
    interface sort
       module procedure sort_int
       module procedure sort_real
    end interface

    interface sort_abs
       module procedure sort_int_abs
    end interface
```

This variant of integer sort returns The body is identical, just the interface differs.

⟨*Sorting: procedures*⟩≡
```
    function sort_int (val_in) result (val)
      integer, dimension(:), intent(in) :: val_in
      integer, dimension(size(val_in)) :: val
    ⟨Sorting: sort⟩
    end function sort_int

    function sort_real (val_in) result (val)
      real(default), dimension(:), intent(in) :: val_in
      real(default), dimension(size(val_in)) :: val
    ⟨Sorting: sort⟩
    end function sort_real
```

```
    function sort_int_abs (val_in) result (val)
      integer, dimension(:), intent(in) :: val_in
      integer, dimension(size(val_in)) :: val
⟨Sorting: sort abs⟩
    end function sort_int_abs
```

⟨Sorting: sort⟩≡
```
    val = val_in( order (val_in) )
```

⟨Sorting: sort abs⟩≡
```
    val = val_in( order_abs (val_in) )
```

The order function returns, for a given integer or real array, the array of indices
of the elements sorted by increasing value.

⟨Sorting: public⟩+≡
```
    public :: order
    public :: order_abs
```

⟨Sorting: interfaces⟩+≡
```
    interface order
       module procedure order_int
       module procedure order_real
    end interface

    interface order_abs
       module procedure order_int_abs
    end interface
```

⟨Sorting: procedures⟩+≡
```
    function order_int (val) result (idx)
      integer, dimension(:), intent(in) :: val
      integer, dimension(size(val)) :: idx
⟨Sorting: order⟩
    end function order_int

    function order_real (val) result (idx)
      real(default), dimension(:), intent(in) :: val
      integer, dimension(size(val)) :: idx
⟨Sorting: order⟩
    end function order_real

    function order_int_abs (val) result (idx)
      integer, dimension(:), intent(in) :: val
      integer, dimension(size(val)) :: idx
⟨Sorting: order abs⟩
    end function order_int_abs
```

We start by individual elements, merge them to pairs, merge those to four-
element subarrays, and so on. The last subarray can extend only up to the
original array bound, of course, and the second of the subarrays to merge should
contain at least one element.

⟨Sorting: order⟩≡
  ⟨Sorting: order1⟩
```
         call merge (idx(b1:e2), idx(b1:e1), idx(b2:e2), val)
```

⟨*Sorting: order2*⟩

⟨*Sorting: order abs*⟩≡
  ⟨*Sorting: order1*⟩
```
        call merge_abs (idx(b1:e2), idx(b1:e1), idx(b2:e2), val)
```
  ⟨*Sorting: order2*⟩

⟨*Sorting: order1*⟩≡
```
    integer :: n, i, s, b1, b2, e1, e2
    n = size (idx)
    forall (i = 1:n)
       idx(i) = i
    end forall
    s = 1
    do while (s < n)
       do b1 = 1, n-s, 2*s
          b2 = b1 + s
          e1 = b2 - 1
          e2 = min (e1 + s, n)
```

⟨*Sorting: order2*⟩≡
```
       end do
       s = 2 * s
    end do
```

The merging step does the actual sorting. We take two sorted array sections
and merge them to a sorted result array. We are working on the indices, and
comparing is done by taking the associated `val` which is real or integer.

⟨*Sorting: interfaces*⟩+≡
```
    interface merge
       module procedure merge_int
       module procedure merge_real
    end interface

    interface merge_abs
       module procedure merge_int_abs
    end interface
```

⟨*Sorting: procedures*⟩+≡
```
    subroutine merge_int (res, src1, src2, val)
      integer, dimension(:), intent(out) :: res
      integer, dimension(:), intent(in) :: src1, src2
      integer, dimension(:), intent(in) :: val
      integer, dimension(size(res)) :: tmp
```
  ⟨*Sorting: merge*⟩
```
    end subroutine merge_int

    subroutine merge_real (res, src1, src2, val)
      integer, dimension(:), intent(out) :: res
      integer, dimension(:), intent(in) :: src1, src2
      real(default), dimension(:), intent(in) :: val
      integer, dimension(size(res)) :: tmp
```
  ⟨*Sorting: merge*⟩
```
    end subroutine merge_real
```

```
      subroutine merge_int_abs (res, src1, src2, val)
        integer, dimension(:), intent(out) :: res
        integer, dimension(:), intent(in) :: src1, src2
        integer, dimension(:), intent(in) :: val
        integer, dimension(size(res)) :: tmp
      ⟨Sorting: merge abs⟩
      end subroutine merge_int_abs
```

⟨*Sorting: merge*⟩≡
  ⟨*Sorting: merge1*⟩
```
        if (val(src1(i1)) <= val(src2(i2))) then
```
  ⟨*Sorting: merge2*⟩

We keep the elements if the absolute values are strictly ordered. If they are
equal in magnitude, we keep them if the larger value comes first, or if they are
equal.

⟨*Sorting: merge abs*⟩≡
  ⟨*Sorting: merge1*⟩
```
        if (abs (val(src1(i1))) < abs (val(src2(i2))) .or. &
            (abs (val(src1(i1))) == abs (val(src2(i2))) .and. &
            val(src1(i1)) >= val(src2(i2)))) then
```
  ⟨*Sorting: merge2*⟩

⟨*Sorting: merge1*⟩≡
```
      integer :: i1, i2, i
      i1 = 1
      i2 = 1
      do i = 1, size (tmp)
```

⟨*Sorting: merge2*⟩≡
```
          tmp(i) = src1(i1);  i1 = i1 + 1
          if (i1 > size (src1)) then
             tmp(i+1:) = src2(i2:)
             exit
          end if
        else
          tmp(i) = src2(i2);  i2 = i2 + 1
          if (i2 > size (src2)) then
             tmp(i+1:) = src1(i1:)
             exit
          end if
        end if
      end do
      res = tmp
```

2.15.2 Concatenating arrays

Not precisely a sorting function, but useful: Concatenate two arrays.

⟨*Sorting: public*⟩+≡
```
      public :: concat
```

⟨*Sorting: interfaces*⟩+≡
```
      interface concat
        module procedure concat_int
        module procedure concat_real
```

```
    end interface
```

⟨*Sorting: procedures*⟩+≡
```
    function concat_int (val1, val2) result (val12)
      integer, dimension(:), intent(in) :: val1, val2
      integer, dimension(size(val1)+size(val2)) :: val12
    ⟨Sorting: concat⟩
    end function concat_int

    function concat_real (val1, val2) result (val12)
      real(default), dimension(:), intent(in) :: val1, val2
      integer, dimension(size(val1)+size(val2)) :: val12
    ⟨Sorting: concat⟩
    end function concat_real
```

⟨*Sorting: concat*⟩≡
```
    val12(:size(val1)) = val1
    val12(size(val1)+1:) = val2
```

2.15.3 Unit tests

⟨*Sorting: public*⟩+≡
```
    public :: sorting_test
```

⟨*Sorting: procedures*⟩+≡
```
    subroutine sorting_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Sorting: execute tests⟩
    end subroutine sorting_test
```

This checks whether the sorting routine works correctly.

⟨*Sorting: execute tests*⟩≡
```
    call test (sorting_1, "sorting_1", &
        "check sorting routines", &
        u, results)
```

⟨*Sorting: tests*⟩≡
```
    subroutine sorting_1 (u)
      integer, intent(in) :: u
      integer, parameter :: NMAX = 10
      real(default), dimension(NMAX) :: rval
      integer, dimension(NMAX) :: ival
      real, dimension(NMAX,NMAX) :: harvest_r
      integer, dimension(NMAX,NMAX) :: harvest_i
      integer, dimension(NMAX,NMAX) :: harvest_a
      integer :: i, j
      harvest_r(:, 1) = [0.9976, 0., 0., 0., 0., 0., 0., 0., 0., 0.]
      harvest_r(:, 2) = [0.5668, 0.9659, 0., 0., 0., 0., 0., 0., 0., 0.]
      harvest_r(:, 3) = [0.7479, 0.3674, 0.4806, 0., 0., 0., 0., 0., 0., &
          0.]
      harvest_r(:, 4) = [0.0738, 0.0054, 0.3471, 0.3422, 0., 0., 0., 0., &
          0., 0.]
```

```
harvest_r(:, 5) = [0.2180, 0.1332, 0.9005, 0.3868, 0.4455, 0., 0., &
     0., 0., 0.]
harvest_r(:, 6) = [0.6619, 0.0161, 0.6509, 0.6464, 0.3230, &
     0.8557, 0., 0., 0., 0.]
harvest_r(:, 7) = [0.4013, 0.2069, 0.9685, 0.5984, 0.6730, &
     0.4569, 0.3300, 0., 0., 0.]
harvest_r(:, 8) = [0.1004, 0.7555, 0.6057, 0.7190, 0.8973, &
     0.6582, 0.1507, 0.6123, 0., 0.]
harvest_r(:, 9) = [0.9787, 0.9991, 0.2568, 0.5509, 0.6590, &
     0.5540, 0.9778, 0.9019, 0.6579, 0.]
harvest_r(:,10) = [0.7289, 0.4025, 0.9286, 0.1478, 0.6745, &
     0.7696, 0.3393, 0.1158, 0.6144, 0.8206]

harvest_i(:, 1) = [18, 0, 0, 0, 0, 0, 0, 0, 0, 0]
harvest_i(:, 2) = [14, 9, 0, 0, 0, 0, 0, 0, 0, 0]
harvest_i(:, 3) = [ 7, 8,11, 0, 0, 0, 0, 0, 0, 0]
harvest_i(:, 4) = [19,19,14,19, 0, 0, 0, 0, 0, 0]
harvest_i(:, 5) = [ 1,14,15,18,14, 0, 0, 0, 0, 0]
harvest_i(:, 6) = [16,11, 1, 9,11, 2, 0, 0, 0, 0]
harvest_i(:, 7) = [11,10,17, 6,13,13,10, 0, 0, 0]
harvest_i(:, 8) = [ 5, 1, 2,10, 7, 0,15,12, 0, 0]
harvest_i(:, 9) = [15,19, 2, 6,11, 0, 2, 4, 2, 0]
harvest_i(:,10) = [ 1, 4, 8, 4,11, 0, 8, 7,19,13]

harvest_a(:, 1) = [-6,  0,  0,  0,  0,  0,  0,  0,  0,  0]
harvest_a(:, 2) = [-8, -9,  0,  0,  0,  0,  0,  0,  0,  0]
harvest_a(:, 3) = [ 4, -3,  3,  0,  0,  0,  0,  0,  0,  0]
harvest_a(:, 4) = [-6,  6,  2, -2,  0,  0,  0,  0,  0,  0]
harvest_a(:, 5) = [ 1, -2,  0, -6,  8,  0,  0,  0,  0,  0]
harvest_a(:, 6) = [-2, -1, -8, -5,  8, -5,  0,  0,  0,  0]
harvest_a(:, 7) = [-9,  0, -6,  2,  5,  3,  2,  0,  0,  0]
harvest_a(:, 8) = [-5, -7,  6,  7, -3,  0, -7,  4,  0,  0]
harvest_a(:, 9) = [ 5,  0, -1, -7,  5,  2,  7, -3,  3,  0]
harvest_a(:,10) = [-9,  2, -6,  3, -9,  5,  5,  7,  5, -9]


write (u, "(A)")  "* Test output: Sorting"
write (u, "(A)")  "*   Purpose: test sorting routines"
write (u, "(A)")

write (u, "(A)")  "* Sorting real values:"

do i = 1, NMAX
   write (u, "(A)")
   rval(:i) = harvest_r(:i,i)
   write (u, "(10(1x,F7.4))") rval(:i)
   rval(:i) = sort (rval(:i))
   write (u, "(10(1x,F7.4))") rval(:i)
   do j = i, 2, -1
      if (rval(j)-rval(j-1) < 0) &
         write (u, "(A)") "*** Sorting failure. ***"
   end do
end do
```

```
   write (u, "(A)")
   write (u, "(A)") "* Sorting integer values:"

   do i = 1, NMAX
      write (u, "(A)")
      ival(:i) = harvest_i(:i,i)
      write (u, "(10(1x,I2))") ival(:i)
      ival(:i) = sort (ival(:i))
      write (u, "(10(1x,I2))") ival(:i)
      do j = i, 2, -1
         if (ival(j)-ival(j-1) < 0) &
            write (u, "(A)")  "*** Sorting failure. ***"
      end do
   end do

   write (u, "(A)")
   write (u, "(A)") "* Sorting integer values by absolute value:"

   do i = 1, NMAX
      write (u, "(A)")
      ival(:i) = harvest_a(:i,i)
      write (u, "(10(1x,I2))") ival(:i)
      ival(:i) = sort_abs (ival(:i))
      write (u, "(10(1x,I2))") ival(:i)
      do j = i, 2, -1
         if (abs(ival(j))-abs(ival(j-1)) < 0 .or. &
               (abs(ival(j))==abs(ival(j-1))) .and. ival(j)>ival(j-1)) &
            write (u, "(A)")  "*** Sorting failure. ***"
      end do
   end do

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: sorting_1"

end subroutine sorting_1
```

Chapter 3

Text handling

`WHIZARD` has to handle complex structures in input (and output) data. Doing this in a generic and transparent way requires a generic lexer and parser. The necessary modules are implemented here:

ifiles Implementation of line-oriented internal files in a more flexible way (linked lists of variable-length strings) than the Fortran builtin features.

lexers Read text and transform it into a token stream.

syntax_rules Define the rules for interpreting tokens, to be used by the parser.

parser Categorize tokens (keyword, string, number etc.) and use a set of syntax rules to transform the input into a parse tree.

3.1 Internal files

The internal files introduced here (`ifile`) are a replacement for the built-in
internal files, which are fixed-size arrays of fixed-length character strings. The
`ifile` type is a doubly-linked list of variable-length character strings with line
numbers.

⟨`ifiles.f90`⟩≡
  ⟨*File header*⟩

  ```
  module ifiles
  ```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use limits, only: EOF !NODEP!
  ```

  ⟨*Standard module head*⟩

  ⟨*Ifiles: public*⟩

  ⟨*Ifiles: types*⟩

  ⟨*Ifiles: interfaces*⟩

  ```
  contains
  ```

  ⟨*Ifiles: subroutines*⟩

  ```
  end module ifiles
  ```

3.1.1 iostat codes

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: EOF = iostat_end,  EOR = iostat_eor
```

3.1.2 The line type

The line entry type is for internal use, it is the list entry to be collected in an
`ifile` object.

⟨*Ifiles: types*⟩≡
```
    type :: line_entry_t
       private
       type(line_entry_t), pointer :: previous => null ()
       type(line_entry_t), pointer :: next => null ()
       type(string_t) :: string
       integer :: index
    end type line_entry_t
```

Create a new list entry, given a varying string as input. The line number and
pointers are not set, these make sense only within an `ifile`.

⟨*Ifiles: subroutines*⟩≡
```
    subroutine line_entry_create (line, string)
```

```
      type(line_entry_t), pointer :: line
      type(string_t), intent(in) :: string
      allocate (line)
      line%string = string
    end subroutine line_entry_create
```

Destroy a single list entry: Since the pointer components should not be deallocated explicitly, just deallocate the object itself.

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine line_entry_destroy (line)
      type(line_entry_t), pointer :: line
      deallocate (line)
    end subroutine line_entry_destroy
```

3.1.3 The ifile type

The internal file is a linked list of line entries.

⟨*Ifiles: public*⟩≡
```
    public :: ifile_t
```

⟨*Ifiles: types*⟩+≡
```
    type :: ifile_t
      private
      type(line_entry_t), pointer :: first => null ()
      type(line_entry_t), pointer :: last => null ()
      integer :: n_lines = 0
    end type ifile_t
```

We need no explicit initializer, but a routine which recursively deallocates the contents may be appropriate. After this, existing line pointers may become undefined, so they should be nullified before the file is destroyed.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_clear
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_clear (ifile)
      type(ifile_t), intent(inout) :: ifile
      type(line_entry_t), pointer :: current
      do while (associated (ifile%first))
         current => ifile%first
         ifile%first => current%next
         call line_entry_destroy (current)
      end do
      nullify (ifile%last)
      ifile%n_lines = 0
    end subroutine ifile_clear
```

The finalizer is just an alias for the above.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_final
```

⟨*Ifiles: interfaces*⟩≡
```
    interface ifile_final
        module procedure ifile_clear
    end interface
```

3.1.4 I/O on ifiles

Fill an ifile from an ordinary external file, i.e., I/O unit. If the ifile is not empty, the old contents will be destroyed. We can read a fixed-length character string, an ISO varying string, an ordinary internal file (character-string array), or from an external unit. In the latter case, lines are appended until EOF is reached. Finally, there is a variant which reads from another ifile, effectively copying it.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_read
```

⟨*Ifiles: interfaces*⟩+≡
```
    interface ifile_read
        module procedure ifile_read_from_string
        module procedure ifile_read_from_char
        module procedure ifile_read_from_unit
        module procedure ifile_read_from_char_array
        module procedure ifile_read_from_ifile
    end interface
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_read_from_string (ifile, string)
      type(ifile_t), intent(inout) :: ifile
      type(string_t), intent(in) :: string
      call ifile_clear (ifile)
      call ifile_append (ifile, string)
    end subroutine ifile_read_from_string

    subroutine ifile_read_from_char (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), intent(in) :: char
      call ifile_clear (ifile)
      call ifile_append (ifile, char)
    end subroutine ifile_read_from_char

    subroutine ifile_read_from_char_array (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), dimension(:), intent(in) :: char
      call ifile_clear (ifile)
      call ifile_append (ifile, char)
    end subroutine ifile_read_from_char_array

    subroutine ifile_read_from_unit (ifile, unit, iostat)
      type(ifile_t), intent(inout) :: ifile
      integer, intent(in) :: unit
      integer, intent(out), optional :: iostat
      call ifile_clear (ifile)
      call ifile_append (ifile, unit, iostat)
    end subroutine ifile_read_from_unit
```

```
    subroutine ifile_read_from_ifile (ifile, ifile_in)
      type(ifile_t), intent(inout) :: ifile
      type(ifile_t), intent(in) :: ifile_in
      call ifile_clear (ifile)
      call ifile_append (ifile, ifile_in)
    end subroutine ifile_read_from_ifile
```

Append to an ifile. The same as reading, but without resetting the ifile. In addition, there is a routine for appending a whole ifile.

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_append
```

⟨*Ifiles: interfaces*⟩+≡
```
    interface ifile_append
       module procedure ifile_append_from_string
       module procedure ifile_append_from_char
       module procedure ifile_append_from_unit
       module procedure ifile_append_from_char_array
       module procedure ifile_append_from_ifile
    end interface
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_append_from_string (ifile, string)
      type(ifile_t), intent(inout) :: ifile
      type(string_t), intent(in) :: string
      type(line_entry_t), pointer :: current
      call line_entry_create (current, string)
      current%index = ifile%n_lines + 1
      if (associated (ifile%last)) then
         current%previous => ifile%last
         ifile%last%next => current
      else
         ifile%first => current
      end if
      ifile%last => current
      ifile%n_lines = current%index
    end subroutine ifile_append_from_string

    subroutine ifile_append_from_char (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), intent(in) :: char
      call ifile_append_from_string (ifile, var_str (trim (char)))
    end subroutine ifile_append_from_char

    subroutine ifile_append_from_char_array (ifile, char)
      type(ifile_t), intent(inout) :: ifile
      character(*), dimension(:), intent(in) :: char
      integer :: i
      do i = 1, size (char)
         call ifile_append_from_string (ifile, var_str (trim (char(i))))
      end do
    end subroutine ifile_append_from_char_array

    subroutine ifile_append_from_unit (ifile, unit, iostat)
      type(ifile_t), intent(inout) :: ifile
```

```
      integer, intent(in) :: unit
      integer, intent(out), optional :: iostat
      type(string_t) :: buffer
      integer :: ios
      ios = 0
      READ_LOOP: do
         call get (unit, buffer, iostat = ios)
         if (ios == EOF .or. ios > 0)  exit READ_LOOP
         call ifile_append_from_string (ifile, buffer)
      end do READ_LOOP
      if (present (iostat)) then
         iostat = ios
      else if (ios > 0) then
         call get (unit, buffer)  ! trigger error again
      end if
   end subroutine ifile_append_from_unit

   subroutine ifile_append_from_ifile (ifile, ifile_in)
     type(ifile_t), intent(inout) :: ifile
     type(ifile_t), intent(in) :: ifile_in
     type(line_entry_t), pointer :: current
     current => ifile_in%first
     do while (associated (current))
        call ifile_append_from_string (ifile, current%string)
        current => current%next
     end do
   end subroutine ifile_append_from_ifile
```

Write the ifile contents to an external unit

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_write
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_write (ifile, unit, iostat)
      type(ifile_t), intent(in) :: ifile
      integer, intent(in), optional :: unit
      integer, intent(out), optional :: iostat
      integer :: u
      type(line_entry_t), pointer :: current
      u = output_unit (unit);  if (u < 0)  return
      current => ifile%first
      do while (associated (current))
         call put_line (u, current%string, iostat)
         current => current%next
      end do
    end subroutine ifile_write
```

Convert the ifile to an array of strings, which is allocated by this function:

⟨*Ifiles: public*⟩+≡
```
    public :: ifile_to_string_array
```

⟨*Ifiles: subroutines*⟩+≡
```
    subroutine ifile_to_string_array (ifile, string)
      type(ifile_t), intent(in) :: ifile
```

```
      type(string_t), dimension(:), intent(inout), allocatable :: string
      type(line_entry_t), pointer :: current
      integer :: i
      allocate (string (ifile_get_length (ifile)))
      current => ifile%first
      do i = 1, ifile_get_length (ifile)
         string(i) = current%string
         current => current%next
      end do
   end subroutine ifile_to_string_array
```

3.1.5 Ifile tools

⟨*Ifiles: public*⟩+≡
```
   public :: ifile_get_length
```

⟨*Ifiles: subroutines*⟩+≡
```
   function ifile_get_length (ifile) result (length)
     integer :: length
     type(ifile_t), intent(in) :: ifile
     length = ifile%n_lines
   end function ifile_get_length
```

3.1.6 Line pointers

Instead of the implicit pointer used in ordinary file access, we define explicit
pointers, so there can be more than one at a time.

⟨*Ifiles: public*⟩+≡
```
   public :: line_p
```

⟨*Ifiles: types*⟩+≡
```
   type :: line_p
     private
     type(line_entry_t), pointer :: p => null ()
   end type line_p
```

Assign a file pointer to the first or last line in an ifile:

⟨*Ifiles: public*⟩+≡
```
   public :: line_init
```

⟨*Ifiles: subroutines*⟩+≡
```
   subroutine line_init (line, ifile, back)
     type(line_p), intent(inout) :: line
     type(ifile_t), intent(in) :: ifile
     logical, intent(in), optional :: back
     if (present (back)) then
        if (back) then
           line%p => ifile%last
        else
           line%p => ifile%first
        end if
     else
```

```
        line%p => ifile%first
      end if
    end subroutine line_init
```

Remove the pointer association:

⟨Ifiles: public⟩+≡
```
    public :: line_final
```

⟨Ifiles: subroutines⟩+≡
```
    subroutine line_final (line)
      type(line_p), intent(inout) :: line
      nullify (line%p)
    end subroutine line_final
```

Go one step forward

⟨Ifiles: public⟩+≡
```
    public :: line_advance
```

⟨Ifiles: subroutines⟩+≡
```
    subroutine line_advance (line)
      type(line_p), intent(inout) :: line
      if (associated (line%p))  line%p => line%p%next
    end subroutine line_advance
```

Go one step backward

⟨Ifiles: public⟩+≡
```
    public :: line_backspace
```

⟨Ifiles: subroutines⟩+≡
```
    subroutine line_backspace (line)
      type(line_p), intent(inout) :: line
      if (associated (line%p))  line%p => line%p%previous
    end subroutine line_backspace
```

Check whether we are accessing a valid line

⟨Ifiles: public⟩+≡
```
    public :: line_is_associated
```

⟨Ifiles: subroutines⟩+≡
```
    function line_is_associated (line) result (ok)
      logical :: ok
      type(line_p), intent(in) :: line
      ok = associated (line%p)
    end function line_is_associated
```

3.1.7   Access lines via pointers

We do not need the ifile as an argument to these functions, because the `line` type will point to an existing ifile.

⟨Ifiles: public⟩+≡
```
    public :: line_get_string
```

⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_string (line) result (string)
      type(string_t) :: string
      type(line_p), intent(in) :: line
      if (associated (line%p)) then
         string = line%p%string
      else
         string = ""
      end if
    end function line_get_string
```

Variant where the line pointer is advanced after reading.
⟨*Ifiles: public*⟩+≡
```
    public :: line_get_string_advance
```
⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_string_advance (line) result (string)
      type(string_t) :: string
      type(line_p), intent(inout) :: line
      if (associated (line%p)) then
         string = line%p%string
         call line_advance (line)
      else
         string = ""
      end if
    end function line_get_string_advance
```

⟨*Ifiles: public*⟩+≡
```
    public :: line_get_index
```
⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_index (line) result (index)
      integer :: index
      type(line_p), intent(in) :: line
      if (associated (line%p)) then
         index = line%p%index
      else
         index = 0
      end if
    end function line_get_index
```

⟨*Ifiles: public*⟩+≡
```
    public :: line_get_length
```
⟨*Ifiles: subroutines*⟩+≡
```
    function line_get_length (line) result (length)
      integer :: length
      type(line_p), intent(in) :: line
      if (associated (line%p)) then
         length = len (line%p%string)
      else
         length = 0
      end if
    end function line_get_length
```

3.2 Lexer

The lexer purpose is to read from a line-separated character input stream (usually a file) and properly chop the stream into lexemes (tokens). [The parser will transform lexemes into meaningful tokens, to be stored in a parse tree, therefore we do not use the term 'token' here.] The input is read line-by-line, but interpreted free-form, except for quotes and the comment syntax. (Fortran 2003 would allow us to use a stream type for reading.)

In an object-oriented approach, we can dynamically create and destroy lexers, including the lexer setup.

The main lexer function is to return a lexeme according to the basic lexer rules (quotes, comments, whitespace, special classes). There is also a routine to write back a lexeme to the input stream (but only once).

For the rules, we separate the possible characters into classes. Whitespace usually consists of blank, tab, and line-feed, where any number of consecutive whitespace is equivalent to one. Quoted strings are enclosed by a pair of quote characters, possibly multiline. Comments are similar to quotes, but interpreted as whitespace. Numbers are identified (not distinguishing real and integer) but not interpreted. Other character classes make up identifiers.

⟨lexers.f90⟩≡
  ⟨File header⟩

  module lexers

  ⟨Use strings⟩
  ⟨Use file utils⟩
    use limits, only: EOF, EOR !NODEP!
    use limits, only: LF !NODEP!
    use limits, only: WHITESPACE_CHARS, LCLETTERS, UCLETTERS, DIGITS !NODEP!
    use diagnostics !NODEP!
    use ifiles, only: ifile_t
    use ifiles, only: line_p, line_is_associated, line_init, line_final
    use ifiles, only: line_get_string_advance
    use unit_tests

  ⟨Standard module head⟩

  ⟨Lexer: public⟩

  ⟨Lexer: parameters⟩

  ⟨Lexer: types⟩

  ⟨Lexer: interfaces⟩

  contains

  ⟨Lexer: procedures⟩

  ⟨Lexer: tests⟩

  end module lexers

3.2.1 Input streams

For flexible input, we define a generic stream type that refers to either an external file, an external unit which is already open, a string, an `ifile` object (internal file, i.e., string list), or a line pointer to an `ifile` object. The stream type actually follows the idea of a formatted external file, which is line-oriented. Thus, the stream reader always returns a whole record (input line).

Note that only in the string version, the stream contents are stored inside the stream object. In the `ifile` version, the stream contains only the line pointer, while in the external-file case, the line pointer is implicitly created by the runtime library.

⟨*Lexer: public*⟩≡
```
    public :: stream_t
```

⟨*Lexer: types*⟩≡
```
    type :: stream_t
       type(string_t), pointer :: filename => null ()
       integer, pointer :: unit => null ()
       type(string_t), pointer :: string => null ()
       type(ifile_t), pointer :: ifile => null ()
       type(line_p), pointer :: line => null ()
       integer :: record = 0
       logical :: eof = .false.
    end type stream_t
```

The initializers refer to the specific version. The stream should be undefined before calling this.

⟨*Lexer: public*⟩+≡
```
    public :: stream_init
```

⟨*Lexer: interfaces*⟩≡
```
    interface stream_init
       module procedure stream_init_filename
       module procedure stream_init_unit
       module procedure stream_init_string
       module procedure stream_init_ifile
       module procedure stream_init_line
    end interface
```

⟨*Lexer: procedures*⟩≡
```
    subroutine stream_init_filename (stream, filename)
      type(stream_t), intent(out) :: stream
      character(*), intent(in) :: filename
      integer :: unit
      unit = free_unit ()
      open (unit=unit, file=filename, status="old", action="read")
      call stream_init_unit (stream, unit)
      allocate (stream%filename)
      stream%filename = filename
    end subroutine stream_init_filename

    subroutine stream_init_unit (stream, unit)
      type(stream_t), intent(out) :: stream
      integer, intent(in) :: unit
```

```
      allocate (stream%unit)
      stream%unit = unit
      stream%eof = .false.
    end subroutine stream_init_unit

    subroutine stream_init_string (stream, string)
      type(stream_t), intent(out) :: stream
      type(string_t), intent(in) :: string
      allocate (stream%string)
      stream%string = string
    end subroutine stream_init_string

    subroutine stream_init_ifile (stream, ifile)
      type(stream_t), intent(out) :: stream
      type(ifile_t), intent(in) :: ifile
      type(line_p) :: line
      call line_init (line, ifile)
      call stream_init_line (stream, line)
      allocate (stream%ifile)
      stream%ifile = ifile
    end subroutine stream_init_ifile

    subroutine stream_init_line (stream, line)
      type(stream_t), intent(out) :: stream
      type(line_p), intent(in) :: line
      allocate (stream%line)
      stream%line = line
    end subroutine stream_init_line
```

The finalizer restores the initial state. If an external file was opened, it is closed.

⟨*Lexer: public*⟩+≡
```
    public :: stream_final
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine stream_final (stream)
      type(stream_t), intent(inout) :: stream
      if (associated (stream%filename)) then
         close (stream%unit)
         deallocate (stream%unit)
         deallocate (stream%filename)
      else if (associated (stream%unit)) then
         deallocate (stream%unit)
      else if (associated (stream%string)) then
         deallocate (stream%string)
      else if (associated (stream%ifile)) then
         call line_final (stream%line)
         deallocate (stream%line)
         deallocate (stream%ifile)
      else if (associated (stream%line)) then
         call line_final (stream%line)
         deallocate (stream%line)
      end if
    end subroutine stream_final
```

This returns the next record from the input stream. Depending on the stream type, the stream pointers are modified: Reading from external unit, the external file is advanced (implicitly). Reading from string, the string is replaced by an empty string. Reading from `ifile`, the line pointer is advanced. Note that the `iostat` argument is mandatory.

⟨*Lexer: public*⟩+≡
```
    public :: stream_get_record
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine stream_get_record (stream, string, iostat)
      type(stream_t), intent(inout) :: stream
      type(string_t), intent(out) :: string
      integer, intent(out) :: iostat
      if (associated (stream%unit)) then
         if (stream%eof) then
            iostat = EOF
         else
            call get (stream%unit, string, iostat=iostat)
            if (iostat == EOR) then
               iostat = 0
               stream%record = stream%record + 1
            end if
            if (iostat == EOF) then
               iostat = 0
               stream%eof = .true.
               if (len (string) /= 0) stream%record = stream%record + 1
            end if
         end if
      else if (associated (stream%string)) then
         if (len (stream%string) /= 0) then
            string = stream%string
            stream%string = ""
            iostat = 0
            stream%record = stream%record + 1
         else
            string = ""
            iostat = EOF
         end if
      else if (associated (stream%line)) then
         if (line_is_associated (stream%line)) then
            string = line_get_string_advance (stream%line)
            iostat = 0
            stream%record = stream%record + 1
         else
            string = ""
            iostat = EOF
         end if
      else
         call msg_bug (" Attempt to read from uninitialized input stream")
      end if
    end subroutine stream_get_record
```

Return the current stream source as a message string.

⟨*Lexer: public*⟩+≡

```
      public :: stream_get_source_info_string
```

⟨*Lexer: procedures*⟩+≡
```
    function stream_get_source_info_string (stream) result (string)
      type(string_t) :: string
      type(stream_t), intent(in) :: stream
      character(20) :: buffer
      if (associated (stream%filename)) then
         string = "File '" // stream%filename // "' (unit = "
         write (buffer, "(I0)")  stream%unit
         string = string // trim (buffer) // ")"
      else if (associated (stream%unit)) then
         write (buffer, "(I0)")  stream%unit
         string = "Unit " // trim (buffer)
      else if (associated (stream%string)) then
         string = "Input string"
      else if (associated (stream%ifile) .or. associated (stream%line)) then
         string = "Internal file"
      else
         string = ""
      end if
    end function stream_get_source_info_string
```

Return the index of the record just read as a message string.

⟨*Lexer: public*⟩+≡
```
      public :: stream_get_record_info_string
```

⟨*Lexer: procedures*⟩+≡
```
    function stream_get_record_info_string (stream) result (string)
      type(string_t) :: string
      type(stream_t), intent(in) :: stream
      character(20) :: buffer
      string = stream_get_source_info_string (stream)
      if (string /= "")  string = string // ", "
      write (buffer, "(I0)")  stream%record
      string = string // "line " // trim (buffer)
    end function stream_get_record_info_string
```

3.2.2 Keyword list

The lexer should be capable of identifying a token as a known keyword. To this
end, we store a list of keywords:

⟨*Lexer: public*⟩+≡
```
      public :: keyword_list_t
```

⟨*Lexer: types*⟩+≡
```
    type :: keyword_entry_t
      private
      type(string_t) :: string
      type(keyword_entry_t), pointer :: next => null ()
    end type keyword_entry_t

    type :: keyword_list_t
```

147

```
    private
    type(keyword_entry_t), pointer :: first => null ()
    type(keyword_entry_t), pointer :: last => null ()
  end type keyword_list_t
```

Add a new string to the keyword list, unless it is already there:

⟨*Lexer: public*⟩+≡
```
    public :: keyword_list_add
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine keyword_list_add (keylist, string)
      type(keyword_list_t), intent(inout) :: keylist
      type(string_t), intent(in) :: string
      type(keyword_entry_t), pointer :: k_entry_new
      if (.not. keyword_list_contains (keylist, string)) then
         allocate (k_entry_new)
         k_entry_new%string = string
         if (associated (keylist%first)) then
            keylist%last%next => k_entry_new
         else
            keylist%first => k_entry_new
         end if
         keylist%last => k_entry_new
      end if
    end subroutine keyword_list_add
```

Return true if a string is a keyword.

⟨*Lexer: public*⟩+≡
```
    public :: keyword_list_contains
```

⟨*Lexer: procedures*⟩+≡
```
    function keyword_list_contains (keylist, string) result (found)
      type(keyword_list_t), intent(in) :: keylist
      type(string_t), intent(in) :: string
      logical :: found
      found = .false.
      call check_rec (keylist%first)
    contains
      recursive subroutine check_rec (k_entry)
        type(keyword_entry_t), pointer :: k_entry
        if (associated (k_entry)) then
           if (k_entry%string /= string) then
              call check_rec (k_entry%next)
           else
              found = .true.
           end if
        end if
      end subroutine check_rec
    end function keyword_list_contains
```

Write the keyword list

⟨*Lexer: public*⟩+≡
```
    public :: keyword_list_write
```

⟨Lexer: interfaces⟩+≡
```
    interface keyword_list_write
       module procedure keyword_list_write_unit
    end interface
```
⟨Lexer: procedures⟩+≡
```
    subroutine keyword_list_write_unit (keylist, unit)
      type(keyword_list_t), intent(in) :: keylist
      integer, intent(in) :: unit
      write (unit, "(A)") "Keyword list:"
      if (associated (keylist%first)) then
         call keyword_write_rec (keylist%first)
         write (unit, *)
      else
         write (unit, "(1x,A)") "[empty]"
      end if
    contains
      recursive subroutine keyword_write_rec (k_entry)
        type(keyword_entry_t), intent(in), pointer :: k_entry
        if (associated (k_entry)) then
           write (unit, "(1x,A)", advance="no")  char (k_entry%string)
           call keyword_write_rec (k_entry%next)
        end if
      end subroutine keyword_write_rec
    end subroutine keyword_list_write_unit
```

Clear the keyword list

⟨Lexer: public⟩+≡
```
    public :: keyword_list_final
```
⟨Lexer: procedures⟩+≡
```
    subroutine keyword_list_final (keylist)
      type(keyword_list_t), intent(inout) :: keylist
      call keyword_destroy_rec (keylist%first)
      nullify (keylist%last)
    contains
      recursive subroutine keyword_destroy_rec (k_entry)
        type(keyword_entry_t), pointer :: k_entry
        if (associated (k_entry)) then
           call keyword_destroy_rec (k_entry%next)
           deallocate (k_entry)
        end if
      end subroutine keyword_destroy_rec
    end subroutine keyword_list_final
```

3.2.3 Lexeme templates

This type is handled like a rudimentary regular expression. It determines the
lexer behavior when matching a string. The actual objects made from this type
and the corresponding matching routines are listed below.

⟨Lexer: types⟩+≡
```
    type :: template_t
```

```
      private
      integer :: type
      character(256) :: charset1, charset2
      integer :: len1, len2
   end type template_t
```

These are the types that valid lexemes can have:

⟨*Lexer: public*⟩+≡
```
   public :: T_KEYWORD, T_IDENTIFIER, T_QUOTED, T_NUMERIC
```

⟨*Lexer: parameters*⟩≡
```
   integer, parameter :: T_KEYWORD = 1
   integer, parameter :: T_IDENTIFIER = 2, T_QUOTED = 3, T_NUMERIC = 4
```

These are special types:

⟨*Lexer: parameters*⟩+≡
```
   integer, parameter :: EMPTY = 0, WHITESPACE = 10
   integer, parameter :: NO_MATCH = 11, IO_ERROR = 12, OVERFLOW = 13
   integer, parameter :: UNMATCHED_QUOTE = 14
```

In addition, we have EOF which is a negative integer, normally -1. Printout for debugging:

⟨*Lexer: procedures*⟩+≡
```
   subroutine lexeme_type_write (type, unit)
     integer, intent(in) :: type
     integer, intent(in) :: unit
     select case (type)
     case (EMPTY);        write(unit,"(A)",advance="no") " EMPTY      "
     case (WHITESPACE);   write(unit,"(A)",advance="no") " WHITESPACE "
     case (T_IDENTIFIER); write(unit,"(A)",advance="no") " IDENTIFIER "
     case (T_QUOTED);     write(unit,"(A)",advance="no") " QUOTED     "
     case (T_NUMERIC);    write(unit,"(A)",advance="no") " NUMERIC    "
     case (IO_ERROR);     write(unit,"(A)",advance="no") " IO_ERROR   "
     case (OVERFLOW);     write(unit,"(A)",advance="no") " OVERFLOW   "
     case (UNMATCHED_QUOTE);   write(unit,"(A)",advance="no") " UNMATCHEDQ "
     case (NO_MATCH);     write(unit,"(A)",advance="no") " NO_MATCH   "
     case (EOF);          write(unit,"(A)",advance="no") " EOF        "
     case default;        write(unit,"(A)",advance="no") " [illegal]  "
     end select
   end subroutine lexeme_type_write

   subroutine template_write (tt, unit)
     type(template_t), intent(in) :: tt
     integer, intent(in) :: unit
     call lexeme_type_write (tt%type, unit)
     write (unit, "(A)", advance="no") "'" // tt%charset1(1:tt%len1) // "'"
     write (unit, "(A)", advance="no") " '" // tt%charset2(1:tt%len2) // "'"
   end subroutine template_write
```

The matching functions all return the number of matched characters in the provided string. If this number is zero, the match has failed.

The template functions are declared pure because they appear in forall loops below.

A template for whitespace:

⟨*Lexer: procedures*⟩+≡

```
pure function template_whitespace (chars) result (tt)
  character(*), intent(in) :: chars
  type(template_t) :: tt
  tt = template_t (WHITESPACE, chars, "", len (chars), 0)
end function template_whitespace
```

Just match the string against the character set.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_whitespace (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  n = verify (s, tt%charset1(1:tt%len1)) - 1
  if (n < 0)  n = len (s)
end subroutine match_whitespace
```

A template for normal identifiers. To match, a lexeme should have a first character in class `chars1` and an arbitrary number of further characters in class `chars2`. If the latter is empty, we are looking for a single-character lexeme.

⟨*Lexer: procedures*⟩+≡

```
pure function template_identifier (chars1, chars2) result (tt)
  character(*), intent(in) :: chars1, chars2
  type(template_t) :: tt
  tt = template_t (T_IDENTIFIER, chars1, chars2, len(chars1), len(chars2))
end function template_identifier
```

Here, the first letter must match, the others may or may not.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_identifier (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  if (verify (s(1:1), tt%charset1(1:tt%len1)) == 0) then
     n = verify (s(2:), tt%charset2(1:tt%len2))
     if (n == 0)  n = len (s)
  else
     n = 0
  end if
end subroutine match_identifier
```

A template for quoted strings. The same template applies for comments. The first character set indicates the left quote (could be a sequence of several characters), the second one the matching right quote.

⟨*Lexer: procedures*⟩+≡

```
pure function template_quoted (chars1, chars2) result (tt)
  character(*), intent(in) :: chars1, chars2
  type(template_t) :: tt
  tt = template_t (T_QUOTED, chars1, chars2, len (chars1), len (chars2))
end function template_quoted
```

Here, the beginning of the string must exactly match the first character set, then we look for the second one. If found, return. If there is a first quote but no second one, return a negative number, indicating this error condition.

⟨*Lexer: procedures*⟩+≡
```
subroutine match_quoted (tt, s, n, range)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  character(tt%len1) :: ch1
  character(tt%len2) :: ch2
  integer :: i
  ch1 = tt%charset1
  if (s(1:tt%len1) == ch1) then
      ch2 = tt%charset2
      do i = tt%len1 + 1, len (s) - tt%len2 + 1
          if (s(i:i+tt%len2-1) == ch2) then
              n = i + tt%len2 - 1
              range(1) = tt%len1 + 1
              range(2) = i - 1
              return
          end if
      end do
      n = -1
      range = 0
  else
      n = 0
      range = 0
  end if
end subroutine match_quoted
```

A template for real numbers. The first character set is the set of allowed exponent letters. In accordance with the other functions we return the lexeme as a string but do not read it.

⟨*Lexer: procedures*⟩+≡
```
pure function template_numeric (chars) result (tt)
  character(*), intent(in) :: chars
  type(template_t) :: tt
  tt = template_t (T_NUMERIC, chars, "", len (chars), 0)
end function template_numeric
```

A numeric lexeme may be real or integer. We purposely do not allow for a preceding sign. If the number is followed by an exponent, this is included, otherwise the rest is ignored.

There is a possible pitfall with this behavior: while the string `1e3` will be interpreted as a single number, the analogous string `1a3` will be split into the number 1 and an identifier `a3`. There is no easy way around such an ambiguity. We should make sure that the syntax does not contain identifiers like `a3` or `e3`.

⟨*Lexer: procedures*⟩+≡
```
subroutine match_numeric (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
```

```fortran
      integer, intent(out) :: n
      integer :: i, n0
      character(10), parameter :: digits = "0123456789"
      character(2), parameter :: signs = "-+"
      n = verify (s, digits) - 1
      if (n < 0) then
         n = 0
         return
      else if (s(n+1:n+1) == ".") then
         i = verify (s(n+2:), digits) - 1
         if (i < 0) then
            n = len (s)
            return
         else if (i > 0 .or. n > 0) then
            n = n + 1 + i
         end if
      end if
      n0 = n
      if (n > 0) then
         if (verify (s(n+1:n+1), tt%charset1(1:tt%len1)) == 0) then
            n = n + 1
            if (verify (s(n+1:n+1), signs) == 0)  n = n + 1
            i = verify (s(n+1:), digits) - 1
            if (i < 0) then
               n = len (s)
            else if (i == 0) then
               n = n0
            else
               n = n + i
            end if
         end if
      end if
   end subroutine match_numeric
```

The generic matching routine. With Fortran 2003 we would define separate types and use a SELECT TYPE instead.

⟨*Lexer: procedures*⟩+≡

```fortran
   subroutine match_template (tt, s, n, range)
     type(template_t), intent(in) :: tt
     character(*), intent(in) :: s
     integer, intent(out) :: n
     integer, dimension(2), intent(out) :: range
     select case (tt%type)
     case (WHITESPACE)
        call match_whitespace (tt, s, n)
        range = 0
     case (T_IDENTIFIER)
        call match_identifier (tt, s, n)
        range(1) = 1
        range(2) = len_trim (s)
     case (T_QUOTED)
        call match_quoted (tt, s, n, range)
     case (T_NUMERIC)
```

```
      call match_numeric (tt, s, n)
      range(1) = 1
      range(2) = len_trim (s)
   case default
      call msg_bug ("Invalid lexeme template encountered")
   end select
end subroutine match_template
```

Match against an array of templates. Return the index of the first template that matches together with the number of characters matched and the range of the relevant substring. If all fails, these numbers are zero.

⟨*Lexer: procedures*⟩+≡
```
subroutine match (tt, s, n, range, ii)
  type(template_t), dimension(:), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  integer, intent(out) :: ii
  integer :: i
  do i = 1, size (tt)
     call match_template (tt(i), s, n, range)
     if (n /= 0) then
        ii = i
        return
     end if
  end do
  n = 0
  ii = 0
end subroutine match
```

3.2.4   The lexer setup

This object contains information about character classes. As said above, one class consists of quoting chars (matching left and right), another one of comment chars (similar), a class of whitespace, and several classes of characters that make up identifiers. When creating the lexer setup, the character classes are transformed into lexeme templates which are to be matched in a certain predefined order against the input stream.

BLANK should always be taken as whitespace, some things may depend on this. TAB is also fixed by convention, but may in principle be modified. Newline (DOS!) and linefeed are also defined as whitespace.

⟨*Limits: public parameters*⟩+≡
```
character, parameter, public :: BLANK = ' ',  TAB = achar(9)
character, parameter, public :: CR = achar(13), LF = achar(10)
character, parameter, public :: BACKSLASH = achar(92)
character(*), parameter, public :: WHITESPACE_CHARS = BLANK// TAB // CR // LF
character(*), parameter, public :: LCLETTERS = "abcdefghijklmnopqrstuvwxyz"
character(*), parameter, public :: UCLETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
character(*), parameter, public :: DIGITS = "0123456789"
```

The lexer setup, containing the list of lexeme templates. No defaults yet. The type with index zero will be assigned to the NO_MATCH lexeme.

The keyword list is not stored, just a pointer to it. We anticipate that the keyword list is part of the syntax table, and the lexer needs not alter it. Furthermore, the lexer is typically finished before the syntax table is.

⟨*Lexer: parameters*⟩+≡
```
integer, parameter :: CASE_KEEP = 0, CASE_UP = 1, CASE_DOWN = 2
```

⟨*Lexer: types*⟩+≡
```
type :: lexer_setup_t
  private
  type(template_t), dimension(:), allocatable :: tt
  integer, dimension(:), allocatable :: type
  integer :: keyword_case = CASE_KEEP
  type(keyword_list_t), pointer :: keyword_list => null ()
end type lexer_setup_t
```

Fill the lexer setup object. Some things are hardcoded here (whitespace, alphanumeric identifiers), some are free: comment chars (but these must be single, and comments must be terminated by line-feed), quote chars and matches (must be single), characters to be read as one-character lexeme, special classes (characters of one class that should be glued together as identifiers).

⟨*Lexer: procedures*⟩+≡
```
subroutine lexer_setup_init (setup, &
    comment_chars, quote_chars, quote_match, &
    single_chars, special_class, &
    keyword_list, upper_case_keywords)
  type(lexer_setup_t), intent(inout) :: setup
  character(*), intent(in) :: comment_chars
  character(*), intent(in) :: quote_chars, quote_match
  character(*), intent(in) :: single_chars
  character(*), dimension(:), intent(in) :: special_class
  type(keyword_list_t), pointer :: keyword_list
  logical, intent(in), optional :: upper_case_keywords
  integer :: n, i
  if (present (upper_case_keywords)) then
     if (upper_case_keywords) then
        setup%keyword_case = CASE_UP
     else
        setup%keyword_case = CASE_DOWN
     end if
  else
     setup%keyword_case = CASE_KEEP
  end if
  n = 1 + len (comment_chars) + len (quote_chars) + 1 &
       + len (single_chars) + size (special_class) + 1
  allocate (setup%tt(n))
  allocate (setup%type(0:n))
  n = 0
  setup%type(n) = NO_MATCH
  n = n + 1
  setup%tt(n) = template_whitespace (WHITESPACE_CHARS)
```

```
    setup%type(n) = EMPTY
    forall (i = 1:len(comment_chars))
        setup%tt(n+i) = template_quoted (comment_chars(i:i), LF)
        setup%type(n+i) = EMPTY
    end forall
    n = n + len (comment_chars)
    forall (i = 1:len(quote_chars))
        setup%tt(n+i) = template_quoted (quote_chars(i:i), quote_match(i:i))
        setup%type(n+i) = T_QUOTED
    end forall
    n = n + len (quote_chars)
    setup%tt(n+1) = template_numeric ("EeDd")
    setup%type(n+1) = T_NUMERIC
    n = n + 1
    forall (i = 1:len (single_chars))
        setup%tt(n+i) = template_identifier (single_chars(i:i), "")
        setup%type(n+i) = T_IDENTIFIER
    end forall
    n = n + len (single_chars)
    forall (i = 1:size (special_class))
        setup%tt(n+i) = template_identifier &
            (trim (special_class(i)), trim (special_class(i)))
        setup%type(n+i) = T_IDENTIFIER
    end forall
    n = n + size (special_class)
    setup%tt(n+1) = template_identifier &
        (LCLETTERS//UCLETTERS, LCLETTERS//DIGITS//"_"//UCLETTERS)
    setup%type(n+1) = T_IDENTIFIER
    n = n + 1
    if (n /= size (setup%tt)) &
        call msg_bug ("Size mismatch in lexer setup")
    setup%keyword_list => keyword_list
  end subroutine lexer_setup_init
```

The destructor is needed only if the object is not itself part of an allocatable array
⟨*Lexer: procedures*⟩+≡
```
  subroutine lexer_setup_final (setup)
    type(lexer_setup_t), intent(inout) :: setup
    deallocate (setup%tt, setup%type)
    setup%keyword_list => null ()
  end subroutine lexer_setup_final
```

For debugging: Write the lexer setup
⟨*Lexer: procedures*⟩+≡
```
  subroutine lexer_setup_write (setup, unit)
    type(lexer_setup_t), intent(in) :: setup
    integer, intent(in) :: unit
    integer :: i
    write (unit, "(A)") "Lexer setup:"
    if (allocated (setup%tt)) then
        do i = 1, size (setup%tt)
            call template_write (setup%tt(i), unit)
```

```
          write (unit, '(A)', advance = "no")  " -> "
          call lexeme_type_write (setup%type(i), unit)
          write (unit, *)
       end do
    else
       write (unit, *) "[empty]"
    end if
    if (associated (setup%keyword_list)) then
       call keyword_list_write (setup%keyword_list, unit)
    end if
 end subroutine lexer_setup_write
```

3.2.5   The lexeme type

An object of this type is returned by the lexer. Apart from the lexeme string,
it gives information about the relevant substring (first and last character index)
and the lexeme type. Interpreting the string is up to the parser.

⟨*Lexer: public*⟩+≡
```
    public :: lexeme_t
```

⟨*Lexer: types*⟩+≡
```
    type :: lexeme_t
       private
       integer :: type = EMPTY
       type(string_t) :: s
       integer :: b = 0, e = 0
    end type lexeme_t
```

Debugging aid:

⟨*Lexer: public*⟩+≡
```
    public :: lexeme_write
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexeme_write (t, unit)
      type(lexeme_t), intent(in) :: t
      integer, intent(in) :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (t%type)
      case (T_KEYWORD)
         write (u, *) "KEYWORD:    '" // char (t%s) // "'"
      case (T_IDENTIFIER)
         write (u, *) "IDENTIFIER: '" // char (t%s) // "'"
      case (T_QUOTED)
         write (u, *) "QUOTED:     '" // char (t%s) // "'"
      case (T_NUMERIC)
         write (u, *) "NUMERIC:    '" // char (t%s) // "'"
      case (UNMATCHED_QUOTE)
         write (u, *) "Unmatched quote: "// char (t%s)
      case (OVERFLOW); write (u, *) "Overflow: "// char (t%s)
      case (EMPTY);    write (u, *) "Empty lexeme"
      case (NO_MATCH); write (u, *) "No match"
      case (IO_ERROR); write (u, *) "IO error"
```

```
      case (EOF);       write (u, *) "EOF"
      case default
         write (u, *) "Error"
      end select
   end subroutine lexeme_write
```

Store string and type in a lexeme. The range determines the beginning and end of the relevant part of the string. Check for a keyword.

⟨*Lexer: procedures*⟩+≡

```
   subroutine lexeme_set (t, keyword_list, s, range, type, keyword_case)
     type(lexeme_t), intent(out) :: t
     type(keyword_list_t), pointer :: keyword_list
     type(string_t), intent(in) :: s
     type(string_t) :: keyword
     integer, dimension(2), intent(in) :: range
     integer, intent(in) :: type
     integer, intent(in), optional :: keyword_case
     t%type = type
     if (present (keyword_case)) then
        select case (keyword_case)
        case (CASE_KEEP);   keyword = s
        case (CASE_UP);     keyword = upper_case (s)
        case (CASE_DOWN);   keyword = lower_case (s)
        end select
     else
        keyword = s
     end if
     if (type == T_IDENTIFIER) then
        if (associated (keyword_list)) then
           if (keyword_list_contains (keyword_list, keyword)) &
                t%type = T_KEYWORD
        end if
     end if
     select case (t%type)
     case (T_KEYWORD);  t%s = keyword
     case default;      t%s = s
     end select
     t%b = range(1)
     t%e = range(2)
   end subroutine lexeme_set

   subroutine lexeme_clear (t)
     type(lexeme_t), intent(out) :: t
     t%type = EMPTY
     t%s = ""
   end subroutine lexeme_clear
```

Retrieve the lexeme string, the relevant part of it, and the type. The last function returns true if there is a break condition reached (error or EOF).

⟨*Lexer: public*⟩+≡

```
   public :: lexeme_get_string
   public :: lexeme_get_contents
   public :: lexeme_get_delimiters
```

```
    public :: lexeme_get_type
```

⟨*Lexer: procedures*⟩+≡
```
    function lexeme_get_string (t) result (s)
      type(string_t) :: s
      type(lexeme_t), intent(in) :: t
      s = t%s
    end function lexeme_get_string

    function lexeme_get_contents (t) result (s)
      type(string_t) :: s
      type(lexeme_t), intent(in) :: t
      s = extract (t%s, t%b, t%e)
    end function lexeme_get_contents

    function lexeme_get_delimiters (t) result (del)
      type(string_t), dimension(2) :: del
      type(lexeme_t), intent(in) :: t
      del(1) = extract (t%s, finish = t%b-1)
      del(2) = extract (t%s, start = t%e+1)
    end function lexeme_get_delimiters

    function lexeme_get_type (t) result (type)
      integer :: type
      type(lexeme_t), intent(in) :: t
      type = t%type
    end function lexeme_get_type
```

Check for a generic break condition (error/eof) and for eof in particular.

⟨*Lexer: public*⟩+≡
```
    public :: lexeme_is_break
    public :: lexeme_is_eof
```

⟨*Lexer: procedures*⟩+≡
```
    function lexeme_is_break (t) result (break)
      logical :: break
      type(lexeme_t), intent(in) :: t
      select case (t%type)
      case (EOF, IO_ERROR, OVERFLOW, NO_MATCH)
         break = .true.
      case default
         break = .false.
      end select
    end function lexeme_is_break

    function lexeme_is_eof (t) result (ok)
      logical :: ok
      type(lexeme_t), intent(in) :: t
      ok = t%type == EOF
    end function lexeme_is_eof
```

3.2.6 The lexer object

We store the current lexeme and the current line. The line buffer is set each time a new line is read from file. The working buffer has one character more, to hold any trailing blank. Pointers to line and column are for debugging, they will be used to make up readable error messages for the parser.

⟨*Lexer: public*⟩+≡

```
public :: lexer_t
```

⟨*Lexer: types*⟩+≡

```
type :: lexer_t
   private
   type(lexer_setup_t) :: setup
   type(stream_t), pointer :: stream => null ()
   type(lexeme_t) :: lexeme
   type(string_t) :: previous_line2
   type(string_t) :: previous_line1
   type(string_t) :: current_line
   integer :: lines_read = 0
   integer :: current_column = 0
   integer :: previous_column = 0
   type(string_t) :: buffer
   type(lexer_t), pointer :: parent => null ()
end type lexer_t
```

Create-setup wrapper

⟨*Lexer: public*⟩+≡

```
public :: lexer_init
```

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_init (lexer, &
     comment_chars, quote_chars, quote_match, &
     single_chars, special_class, &
     keyword_list, upper_case_keywords, &
     parent)
  type(lexer_t), intent(inout) :: lexer
  character(*), intent(in) :: comment_chars
  character(*), intent(in) :: quote_chars, quote_match
  character(*), intent(in) :: single_chars
  character(*), dimension(:), intent(in) :: special_class
  type(keyword_list_t), pointer :: keyword_list
  logical, intent(in), optional :: upper_case_keywords
  type(lexer_t), target, intent(in), optional :: parent
  call lexer_setup_init (lexer%setup, &
       comment_chars = comment_chars, &
       quote_chars = quote_chars, &
       quote_match = quote_match, &
       single_chars = single_chars, &
       special_class = special_class, &
       keyword_list = keyword_list, &
       upper_case_keywords = upper_case_keywords)
  if (present (parent))  lexer%parent => parent
  call lexer_clear (lexer)
end subroutine lexer_init
```

Clear the lexer state, but not the setup. This should be done when the lexing starts, but it is not known whether the lexer was used before.

⟨*Lexer: public*⟩+≡
```
    public :: lexer_clear
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_clear (lexer)
      type(lexer_t), intent(inout) :: lexer
      call lexeme_clear (lexer%lexeme)
      lexer%previous_line2 = ""
      lexer%previous_line1 = ""
      lexer%current_line = ""
      lexer%lines_read = 0
      lexer%current_column = 0
      lexer%previous_column = 0
      lexer%buffer = ""
    end subroutine lexer_clear
```

Reset lexer state and delete setup

⟨*Lexer: public*⟩+≡
```
    public :: lexer_final
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_final (lexer)
      type(lexer_t), intent(inout) :: lexer
      call lexer_clear (lexer)
      call lexer_setup_final (lexer%setup)
    end subroutine lexer_final
```

3.2.7 The lexer routine

For lexing we need to associate an input stream to the lexer.

⟨*Lexer: public*⟩+≡
```
    public :: lexer_assign_stream
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_assign_stream (lexer, stream)
      type(lexer_t), intent(inout) :: lexer
      type(stream_t), intent(in), target :: stream
      lexer%stream => stream
    end subroutine lexer_assign_stream
```

The lexer. The lexer function takes the lexer and returns the currently stored lexeme. If there is none, it is read from buffer, matching against the lexeme templates in the lexer setup. Empty lexemes, i.e., comments and whitespace, are discarded and the buffer is read again until we have found a nonempty lexeme (which may also be EOF or an error condition).

The initial state of the lexer contains an empty lexeme, so reading from buffer is forced. The empty state is restored after returning the lexeme. A nonempty lexeme is present in the lexer only if `lex_back` has been executed before.

The workspace is the `lexer%buffer`, treated as a sort of input stream. We chop off lexemes from the beginning, adjusting the buffer to the left. Whenever the buffer is empty, or we are matching against an open quote which has not terminated, we read a new line and append it to the right. This may result in special conditions, which for simplicity are also returned as lexemes: I/O error, buffer overflow, end of file. If the latter happens during reading a quoted string, we return an unmatched-quote lexeme. Obviously, the special-condition lexemes have to be caught by the parser.

Note that reading further lines is only necessary when reading a quoted string. Otherwise, the line-feed that ends each line is interpreted as whitespace which terminates a preceding lexeme, so there are no other valid multiline lexemes.

To enable meaningful error messages, we also keep track of the line number of the last line read, and the beginning and the end of the current lexeme with respect to this line.

The lexer is implemented as a function that returns the next lexeme (i.e., token). It uses the `lexer` setup and modifies the buffers and pointers stored within the lexer, a side effect. The lexer reads from an input stream object, which also is modified by this reading, e.g., a line pointer is advanced.

⟨*Lexer: public*⟩+≡
```
    public :: lex
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lex (lexeme, lexer)
      type(lexeme_t), intent(out) :: lexeme
      type(lexer_t), intent(inout) :: lexer
      integer :: iostat1, iostat2
      integer :: pos
      integer, dimension(2) :: range
      integer :: template_index, type
      if (.not. associated (lexer%stream)) &
          call msg_bug ("Lexer called without assigned stream")
      GET_LEXEME: do while (lexeme_get_type (lexer%lexeme) == EMPTY)
        if (len (lexer%buffer) /= 0) then
          iostat1 = 0
        else
          call lexer_read_line (lexer, iostat1)
        end if
        select case (iostat1)
        case (0)
          MATCH_BUFFER: do
            call match (lexer%setup%tt, char (lexer%buffer), &
                        pos, range, template_index)
            if (pos >= 0) then
              type = lexer%setup%type(template_index)
              exit MATCH_BUFFER
            else
              pos = 0
              call lexer_read_line (lexer, iostat2)
              select case (iostat2)
              case (EOF); type = UNMATCHED_QUOTE; exit MATCH_BUFFER
              case (1);   type = IO_ERROR;        exit MATCH_BUFFER
              case (2);   type = OVERFLOW;        exit MATCH_BUFFER
```

```
                end select
            end if
          end do MATCH_BUFFER
        case (EOF); type = EOF
        case (1);    type = IO_ERROR
        case (2);    type = OVERFLOW
        end select
        call lexeme_set (lexer%lexeme, lexer%setup%keyword_list, &
             extract (lexer%buffer, finish=pos), range, type, &
             lexer%setup%keyword_case)
        lexer%buffer = remove (lexer%buffer, finish=pos)
        lexer%previous_column = lexer%current_column
        lexer%current_column = lexer%current_column + pos
      end do GET_LEXEME
      lexeme = lexer%lexeme
      call lexeme_clear (lexer%lexeme)
    end subroutine lex
```

Read a line and append it to the input buffer. If the input buffer overflows, return `iostat=2`. Otherwise, `iostat=1` indicates an I/O error, and `iostat=-1` the EOF.

The input stream may either be an external unit or a `ifile` object. In the latter case, a line is read and the line pointer is advanced.

Note that inserting `LF` between input lines is the Unix convention. Since we are doing this explicitly when gluing lines together, we can pattern-match against `LF` without having to worry about the system.

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_read_line (lexer, iostat)
      type(lexer_t), intent(inout) :: lexer
      integer, intent(out) :: iostat
      type(string_t) :: current_line
      current_line = lexer%current_line
      call stream_get_record (lexer%stream, lexer%current_line, iostat)
      if (iostat == 0) then
         lexer%lines_read = lexer%lines_read + 1
         lexer%previous_line2 = lexer%previous_line1
         lexer%previous_line1 = current_line
         lexer%buffer = lexer%buffer // lexer%current_line // LF
         lexer%previous_column = 0
         lexer%current_column = 0
      end if
    end subroutine lexer_read_line
```

Once in a while we have read one lexeme to many, which can be pushed back into the input stream. Do not do this more than once.

⟨*Lexer: public*⟩+≡
```
    public :: lexer_put_back
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_put_back (lexer, lexeme)
      type(lexer_t), intent(inout) :: lexer
      type(lexeme_t), intent(in) :: lexeme
      if (lexeme_get_type (lexer%lexeme) == EMPTY) then
```

163

```
        lexer%lexeme = lexeme
     else
        call msg_bug (" Lexer: lex_back fails; probably called twice")
     end if
  end subroutine lexer_put_back
```

3.2.8   Diagnostics

For debugging: print just the setup

⟨*Lexer: public*⟩+≡
```
    public :: lexer_write_setup
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_write_setup (lexer, unit)
      type(lexer_t), intent(in) :: lexer
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call lexer_setup_write (lexer%setup, u)
    end subroutine lexer_write_setup
```

This is useful for error printing: show the current line with index and a pointer
to the current column within the line.

⟨*Lexer: public*⟩+≡
```
    public :: lexer_show_location
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_show_location (lexer)
      type(lexer_t), intent(in) :: lexer
      type(string_t) :: loc_str
      if (associated (lexer%parent)) then
         call lexer_show_source (lexer%parent)
         call msg_message ("[includes]")
      else
         call msg_message ()
      end if
      if (associated (lexer%stream)) then
         call msg_message &
              (char (stream_get_record_info_string (lexer%stream)) // ":")
      end if
      if (lexer%lines_read >= 4)  call msg_result ("[...]")
      if (lexer%lines_read >= 3)  call msg_result (char (lexer%previous_line2))
      if (lexer%lines_read >= 2)  call msg_result (char (lexer%previous_line1))
      if (lexer%lines_read >= 1) then
         call msg_result (char (lexer%current_line))
         loc_str = repeat (" ", lexer%previous_column)
         loc_str = loc_str // "^"
         if (lexer%current_column > lexer%previous_column) then
            loc_str = loc_str &
                 // repeat ("-", max (lexer%current_column &
                                       - lexer%previous_column - 1, 0)) &
                 // "^"
         end if
```

164

```
          call msg_result (char (loc_str))
      end if
    end subroutine lexer_show_location
```

This just prints the current stream source.

⟨*Lexer: procedures*⟩+≡
```
    recursive subroutine lexer_show_source (lexer)
      type(lexer_t), intent(in) :: lexer
      if (associated (lexer%parent)) then
         call lexer_show_source (lexer%parent)
         call msg_message ("[includes]")
      else
         call msg_message ()
      end if
      if (associated (lexer%stream)) then
         call msg_message &
              (char (stream_get_source_info_string (lexer%stream)) // ":")
      end if
    end subroutine lexer_show_source
```

3.2.9   Unit tests

⟨*Lexer: public*⟩+≡
```
    public :: lexer_test
```

⟨*Lexer: procedures*⟩+≡
```
    subroutine lexer_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Lexer: execute tests*⟩
```
    end subroutine lexer_test
```

Test the lexer by lexing and printing all lexemes from unit u, one per line, using
preset conventions.

⟨*Lexer: execute tests*⟩≡
```
    call test (lexer_1, "lexer_1", &
         "check lexer", u, results)
```

⟨*Lexer: tests*⟩≡
```
    subroutine lexer_1 (u)
      integer, intent(in) :: u
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      type(string_t) :: string
      type(lexeme_t) :: lexeme
      string = "abcdefghij"
      call lexer_init (lexer, &
         comment_chars = "", &
         quote_chars = "<'""", &
         quote_match = ">'""", &
         single_chars = "?*+|=,()", &
         special_class = ["."], &
```

```
      keyword_list = null ())
   call stream_init (stream, string)
   call lexer_assign_stream (lexer, stream)
   do
      call lex (lexeme, lexer)
      call lexeme_write (lexeme, u)
      if (lexeme_is_break (lexeme))  exit
   end do
   call stream_final (stream)
   call lexer_final (lexer)
end subroutine lexer_1
```

3.3 Syntax rules

This module provides tools to handle syntax rules in an abstract way.

⟨`syntax_rules.f90`⟩≡
  ⟨*File header*⟩

```
  module syntax_rules
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use limits, only: UNQUOTED !NODEP!
    use diagnostics !NODEP!
    use ifiles, only: line_p, line_init, line_get_string_advance, line_final
    use ifiles, only: ifile_t, ifile_get_length
    use lexers
```

  ⟨*Standard module head*⟩

  ⟨*Syntax: public*⟩

  ⟨*Syntax: parameters*⟩

  ⟨*Syntax: types*⟩

  ⟨*Syntax: interfaces*⟩

```
  contains
```

  ⟨*Syntax: subroutines*⟩

```
  end module syntax_rules
```

3.3.1 Syntax rules

Syntax rules are used by the parser. They determine how to translate the stream of lexemes as returned by the lexer into the parse tree node. A rule may be terminal, i.e., replace a matching lexeme into a terminal node. The node will contain the lexeme interpreted as a recognized token:

- a keyword: unquoted fixed character string;

- a real number, to be determined at runtime;

- an integer, to be determined at runtime;

- a boolean value, to be determined at runtime;

- a quoted token (e.g., string), to be determined at runtime;

- an identifier (unquoted string that is not a recognized keyword), to be determined at runtime.

It may be nonterminal, i.e., contain a sequence of child rules. These are matched consecutively (and recursively) against the input stream; the resulting node will be a branch node.

- the file, i.e., the input stream as a whole;

- a sequence of syntax elements, where the last syntax element may be optional, or optional repetitive;

Sequences carry a flag that tells whether the last child is optional or may be repeated an arbitrary number of times, correponding to the regexp modifiers ?, *, and +.

We also need an alternative rule; this will be replaced by the node generated by one of its children that matches; thus, it does not create a node of its own.

- an alternative of syntax elements.

We also define special types of sequences as convenience macros:

- a list: a sequence where the elements are separated by a separator keyword (e.g., commas), the separators are thrown away when parsing the list;

- a group: a sequence of three tokens, where the first and third ones are left and right delimiters, the delimiters are thrown away;

- an argument list: a delimited list, containing both delimiters and separators.

It would be great to have a polymorphic type for this purpose, but until Fortran 2003 is out we have to emulate this.

Here are the syntax element codes:

⟨*Syntax: public*⟩≡
```
    public :: S_UNKNOWN
    public :: S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_QUOTED
    public :: S_IDENTIFIER, S_KEYWORD
    public :: S_SEQUENCE, S_LIST, S_GROUP, S_ARGS
    public :: S_ALTERNATIVE
    public :: S_IGNORE
```

⟨*Syntax: parameters*⟩≡
```
    integer, parameter :: &
        S_UNKNOWN = 0, &
        S_LOGICAL = 1, S_INTEGER = 2, S_REAL = 3, S_COMPLEX = 4, &
        S_QUOTED = 5, S_IDENTIFIER = 6, S_KEYWORD = 7, &
        S_SEQUENCE = 8, S_LIST = 9, S_GROUP = 10, S_ARGS = 11, &
        S_ALTERNATIVE = 12, &
        S_IGNORE = 99
```

We need arrays of rule pointers, therefore this construct.

⟨*Syntax: types*⟩≡
```
    type :: rule_p
       private
       type(syntax_rule_t), pointer :: p => null ()
    end type rule_p
```

Return the association status of the rule pointer:

⟨*Syntax: subroutines*⟩≡
```
    elemental function rule_is_associated (rp) result (ok)
      logical :: ok
      type (rule_p), intent(in) :: rp
      ok = associated (rp%p)
    end function rule_is_associated
```

The rule type is one of the types listed above, represented by an integer code. The keyword, for a non-keyword rule, is an identifier used for the printed syntax table. The array of children is needed for nonterminal rules. In that case, there is a modifier for the last element (blank, "?", "*", or "+"), mirrored in the flags `opt` and `rep`. Then, we have the character constants used as separators and delimiters for this rule. Finally, the `used` flag can be set to indicate that this rule is the child of another rule.

⟨*Syntax: types*⟩+≡
```
    public :: syntax_rule_t
```

⟨*Syntax: types*⟩+≡
```
    type :: syntax_rule_t
      private
      integer :: type = S_UNKNOWN
      logical :: used = .false.
      type(string_t) :: keyword
      type(string_t) :: separator
      type(string_t), dimension(2) :: delimiter
      type(rule_p), dimension(:), allocatable :: child
      character(1) :: modifier = ""
      logical :: opt = .false., rep = .false.
    end type syntax_rule_t
```

Initializer: Set type and key for a rule, but do not (yet) allocate anything. Finalizer: not needed (no pointer components).

⟨*Syntax: subroutines*⟩+≡
```
    subroutine syntax_rule_init (rule, key, type)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), intent(in) :: key
      integer, intent(in) :: type
      rule%keyword = key
      rule%type = type
      select case (rule%type)
      case (S_GROUP)
         call syntax_rule_set_delimiter (rule)
      case (S_LIST)
         call syntax_rule_set_separator (rule)
      case (S_ARGS)
         call syntax_rule_set_delimiter (rule)
         call syntax_rule_set_separator (rule)
      end select
    end subroutine syntax_rule_init
```

3.3.2 I/O

These characters will not be enclosed in quotes when writing syntax rules:

⟨*Limits: public parameters*⟩+≡
```
character(*), parameter, public :: &
        UNQUOTED = "(),|_"//LCLETTERS//UCLETTERS//DIGITS
```

Write an account of the rule. Setting `short` true will suppress the node type. Setting `key_only` true will suppress the definition. Setting `advance` false will suppress the trailing newline.

⟨*Syntax: public*⟩+≡
```
public :: syntax_rule_write
```

⟨*Syntax: subroutines*⟩+≡
```
subroutine syntax_rule_write (rule, unit, short, key_only, advance)
  type(syntax_rule_t), intent(in) :: rule
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: short, key_only, advance
  logical :: typ, def, adv
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  typ = .true.;  if (present (short))    typ = .not. short
  def = .true.;  if (present (key_only)) def = .not. key_only
  adv = .true.;  if (present (advance))  adv = advance
  select case (rule%type)
  case (S_UNKNOWN);    call write_atom ("???", typ)
  case (S_IGNORE);     call write_atom ("IGNORE", typ)
  case (S_LOGICAL);    call write_atom ("LOGICAL", typ)
  case (S_INTEGER);    call write_atom ("INTEGER", typ)
  case (S_REAL);       call write_atom ("REAL", typ)
  case (S_COMPLEX);    call write_atom ("COMPLEX", typ)
  case (S_IDENTIFIER); call write_atom ("IDENTIFIER", typ)
  case (S_KEYWORD);    call write_atom ("KEYWORD", typ)
  case (S_QUOTED)
     call write_quotes (typ, def, del = rule%delimiter)
  case (S_SEQUENCE)
     call write_sequence ("SEQUENCE", typ, def, size (rule%child))
  case (S_GROUP)
     call write_sequence ("GROUP", typ, def, size (rule%child), &
          del = rule%delimiter)
  case (S_LIST)
     call write_sequence ("LIST", typ, def, size (rule%child), &
          sep = rule%separator)
  case (S_ARGS)
     call write_sequence ("ARGUMENTS", typ, def, size (rule%child), &
          del = rule%delimiter, sep = rule%separator)
  case (S_ALTERNATIVE)
     call write_sequence ("ALTERNATIVE", typ, def, size (rule%child), &
          sep = var_str ("|"))
  end select
  if (adv)  write (u, *)
contains
  subroutine write_type (type)
    character(*), intent(in) :: type
    character(11) :: str
```

```
    str = type
    write (u, "(1x,A)", advance="no")  str
end subroutine write_type
subroutine write_key
    write (u, "(1x,A)", advance="no")  char (wkey (rule))
end subroutine write_key
subroutine write_atom (type, typ)
    character(*), intent(in) :: type
    logical, intent(in) :: typ
    if (typ)  call write_type (type)
    call write_key
end subroutine write_atom
subroutine write_maybe_quoted (string)
    character(*), intent(in) :: string
    character, parameter :: q = "'"
    character, parameter :: qq = '"'
    if (verify (string, UNQUOTED) == 0) then
       write (u, "(1x,A)", advance = "no") trim (string)
    else if (verify (string, q) == 0) then
       write (u, "(1x,A)", advance = "no") qq // trim (string) // qq
    else
       write (u, "(1x,A)", advance = "no") q // trim (string) // q
    end if
end subroutine write_maybe_quoted
subroutine write_quotes (typ, def, del)
    logical, intent(in) :: typ, def
    type(string_t), dimension(2), intent(in) :: del
    if (typ)  call write_type ("QUOTED")
    call write_key
    if (def) then
       write (u, "(1x,'=')", advance="no")
       call write_maybe_quoted (char (del(1)))
       write (u, "(1x,A)", advance="no") "..."
       call write_maybe_quoted (char (del(2)))
    end if
end subroutine write_quotes
subroutine write_sequence (type, typ, def, n, del, sep)
    character(*), intent(in) :: type
    logical, intent(in) :: typ, def
    integer, intent(in) :: n
    type(string_t), dimension(2), intent(in), optional :: del
    type(string_t), intent(in), optional :: sep
    integer :: i
    if (typ)  call write_type (type)
    call write_key
    if (def) then
       write (u, "(1x,'=')", advance="no")
       if (present (del))  call write_maybe_quoted (char (del(1)))
       do i = 1, n
          if (i > 1 .and. present (sep)) &
               call write_maybe_quoted (char (sep))
          write (u, "(1x,A)", advance="no")  &
               char (wkey (syntax_rule_get_sub_ptr(rule, i)))
          if (i == n)  write (u, "(A)", advance="no")  trim (rule%modifier)
```

```
        end do
        if (present (del))  call write_maybe_quoted (char (del(2)))
      end if
    end subroutine write_sequence
  end subroutine syntax_rule_write
```

In the printed representation, the keyword strings are enclosed as <...>, unless they are bare keywords. Bare keywords are enclosed as '..' if they contain a character which is not a letter, digit, or underscore. If they contain a single-quote character, they are enclosed as "..". (A keyword must not contain both single- and double-quotes.)

⟨*Syntax: subroutines*⟩+≡
```
    function wkey (rule) result (string)
      type(string_t) :: string
      type(syntax_rule_t), intent(in) :: rule
      select case (rule%type)
      case (S_KEYWORD)
        if (verify (rule%keyword, UNQUOTED) == 0) then
          string = rule%keyword
        else if (scan (rule%keyword, "'") == 0) then
          string = "'" // rule%keyword // "'"
        else
          string = '"' // rule%keyword // '"'
        end if
      case default
        string = "<" // rule%keyword // ">"
      end select
    end function wkey
```

3.3.3   Completing syntax rules

Set the separator and delimiter entries, using defaults:

⟨*Syntax: subroutines*⟩+≡
```
    subroutine syntax_rule_set_separator (rule, separator)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), intent(in), optional :: separator
      if (present (separator)) then
        rule%separator = separator
      else
        rule%separator = ","
      end if
    end subroutine syntax_rule_set_separator

    subroutine syntax_rule_set_delimiter (rule, delimiter)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), dimension(2), intent(in), optional :: delimiter
      if (present (delimiter)) then
        rule%delimiter = delimiter
      else
        rule%delimiter = ["(", ")"]
      end if
```

```
        end subroutine syntax_rule_set_delimiter
```

Set the modifier entry and corresponding flags:

⟨*Syntax: subroutines*⟩+≡
```
    function is_modifier (string) result (ok)
      logical :: ok
      type(string_t), intent(in) :: string
      select case (char (string))
      case (" ", "?", "*", "+");  ok = .true.
      case default;               ok = .false.
      end select
    end function is_modifier

    subroutine syntax_rule_set_modifier (rule, modifier)
      type(syntax_rule_t), intent(inout) :: rule
      type(string_t), intent(in) :: modifier
      rule%modifier = char (modifier)
      select case (rule%modifier)
      case (" ")
      case ("?");  rule%opt = .true.
      case ("*");  rule%opt = .true.;  rule%rep = .true.
      case ("+");  rule%rep = .true.
      case default
         call msg_bug (" Syntax: sequence modifier '" // rule%modifier &
              // "' is not one of '+' '*' '?'")
      end select
    end subroutine syntax_rule_set_modifier
```

Check a finalized rule for completeness

⟨*Syntax: subroutines*⟩+≡
```
    subroutine syntax_rule_check (rule)
      type(syntax_rule_t), intent(in) :: rule
      if (rule%keyword == "")  call msg_bug ("Rule key not set")
      select case (rule%type)
      case (S_UNKNOWN);  call bug (" Undefined rule")
      case (S_IGNORE, S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, &
               S_IDENTIFIER, S_KEYWORD)
      case (S_QUOTED)
         if (any (rule%delimiter == "")) call bug (" Missing quote character(s)")
      case (S_SEQUENCE)
      case (S_GROUP)
         if (any (rule%delimiter == "")) call bug (" Missing delimiter(s)")
      case (S_LIST)
         if (rule%separator == "") call bug (" Missing separator")
      case (S_ARGS)
         if (any (rule%delimiter == "")) call bug (" Missing delimiter(s)")
         if (rule%separator == "") call bug (" Missing separator")
      case (S_ALTERNATIVE)
      case default
         call bug (" Undefined syntax code")
      end select
      select case (rule%type)
      case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
```

```
            if (allocated (rule%child)) then
               if (.not.all (rule_is_associated (rule%child))) &
                    call bug (" Child rules not all associated")
            else
               call bug (" Parent rule without children")
            end if
         case default
            if (allocated (rule%child))  call bug (" Non-parent rule with children")
         end select
      contains
        subroutine bug (string)
          character(*), intent(in) :: string
          call msg_bug (" Syntax table: Rule " // char (rule%keyword) // ": " &
               // string)
        end subroutine bug
      end subroutine syntax_rule_check
```

3.3.4 Accessing rules

This is the API for syntax rules:

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_type
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_type (rule) result (type)
      integer :: type
      type(syntax_rule_t), intent(in) :: rule
      type = rule%type
    end function syntax_rule_get_type
```

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_key
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_key (rule) result (key)
      type(string_t) :: key
      type(syntax_rule_t), intent(in) :: rule
      key = rule%keyword
    end function syntax_rule_get_key
```

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_get_separator
    public :: syntax_rule_get_delimiter
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_get_separator (rule) result (separator)
      type(string_t) :: separator
      type(syntax_rule_t), intent(in) :: rule
      separator = rule%separator
    end function syntax_rule_get_separator

    function syntax_rule_get_delimiter (rule) result (delimiter)
      type(string_t), dimension(2) :: delimiter
```

```
        type(syntax_rule_t), intent(in) :: rule
        delimiter = rule%delimiter
      end function syntax_rule_get_delimiter
```

Accessing child rules. If we use `syntax_rule_get_n_sub` for determining loop
bounds, we do not need a check in the second routine.

⟨*Syntax: public*⟩+≡
```
      public :: syntax_rule_get_n_sub
      public :: syntax_rule_get_sub_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
      function syntax_rule_get_n_sub (rule) result (n)
        integer :: n
        type(syntax_rule_t), intent(in) :: rule
        if (allocated (rule%child)) then
           n = size (rule%child)
        else
           n = 0
        end if
      end function syntax_rule_get_n_sub

      function syntax_rule_get_sub_ptr (rule, i) result (sub)
        type(syntax_rule_t), pointer :: sub
        type(syntax_rule_t), intent(in), target :: rule
        integer, intent(in) :: i
        sub => rule%child(i)%p
      end function syntax_rule_get_sub_ptr

      subroutine syntax_rule_set_sub (rule, i, sub)
        type(syntax_rule_t), intent(inout) :: rule
        integer, intent(in) :: i
        type(syntax_rule_t), intent(in), target :: sub
        rule%child(i)%p => sub
      end subroutine syntax_rule_set_sub
```

Return the modifier flags:

⟨*Syntax: public*⟩+≡
```
      public :: syntax_rule_last_optional
      public :: syntax_rule_last_repetitive
```

⟨*Syntax: subroutines*⟩+≡
```
      function syntax_rule_last_optional (rule) result (opt)
        logical :: opt
        type(syntax_rule_t), intent(in) :: rule
        opt = rule%opt
      end function syntax_rule_last_optional
      function syntax_rule_last_repetitive (rule) result (rep)
        logical :: rep
        type(syntax_rule_t), intent(in) :: rule
        rep = rule%rep
      end function syntax_rule_last_repetitive
```

Return true if the rule is atomic, i.e., logical, real, keyword etc.

⟨*Syntax: public*⟩+≡
```
    public :: syntax_rule_is_atomic
```

⟨*Syntax: subroutines*⟩+≡
```
    function syntax_rule_is_atomic (rule) result (atomic)
      logical :: atomic
      type(syntax_rule_t), intent(in) :: rule
      select case (rule%type)
      case (S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_IDENTIFIER, &
              S_KEYWORD, S_QUOTED)
         atomic = .true.
      case default
         atomic = .false.
      end select
    end function syntax_rule_is_atomic
```

3.3.5 Syntax tables

A syntax table contains the tree of syntax rules and, for direct parser access, the list of valid keywords.

Types

The syntax contains an array of rules and a list of keywords. The array is actually used as a tree, where the top rule is the first array element, and the other rules are recursively pointed to by this first rule. (No rule should be used twice or be unused.) The keyword list is derived from the rule tree.

Objects of this type need the target attribute if they are associated with a lexer. The keyword list will be pointed to by this lexer.

⟨*Syntax: public*⟩+≡
```
    public :: syntax_t
```

⟨*Syntax: types*⟩+≡
```
    type :: syntax_t
      private
      type(syntax_rule_t), dimension(:), allocatable :: rule
      type(keyword_list_t) :: keyword_list
    end type syntax_t
```

Constructor/destructor

Initialize and finalize syntax tables

⟨*Syntax: public*⟩+≡
```
    public :: syntax_init
    public :: syntax_final
```

There are two ways to create a syntax: hard-coded from rules or dynamically from file.

⟨*Syntax: interfaces*⟩≡
```
interface syntax_init
   module procedure syntax_init_from_ifile
end interface
```

The syntax definition is read from an `ifile` object which contains the syntax definitions in textual form, one rule per line. This interface allows for determining the number of rules beforehand.

To parse the rule definitions, we make up a temporary lexer. Obviously, we cannot use a generic parser yet, so we have to hardcode the parsing process.

⟨*Syntax: subroutines*⟩+≡
```
subroutine syntax_init_from_ifile (syntax, ifile)
  type(syntax_t), intent(out), target :: syntax
  type(ifile_t), intent(in) :: ifile
  type(lexer_t) :: lexer
  type(line_p) :: line
  type(string_t) :: string
  integer :: n_token
  integer :: i
  call lexer_init (lexer, &
     comment_chars = "", &
     quote_chars = "<'""", &
     quote_match = ">'""", &
     single_chars = "?*+|=,()", &
     special_class = ["."], &
     keyword_list = null ())
  allocate (syntax%rule (ifile_get_length (ifile)))
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
     string = line_get_string_advance (line)
     call set_rule_type_and_key (syntax%rule(i), string, lexer)
  end do
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
     string = line_get_string_advance (line)
     select case (syntax%rule(i)%type)
     case (S_QUOTED, S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
        n_token = get_n_token (string, lexer)
        call set_rule_contents &
             (syntax%rule(i), syntax, n_token, string, lexer)
     end select
  end do
  call line_final (line)
  call lexer_final (lexer)
  call syntax_make_keyword_list (syntax)
  if (.not. all (syntax%rule%used)) then
     do i = 1, size (syntax%rule)
        if (.not. syntax%rule(i)%used) then
           call syntax_rule_write (syntax%rule(i), 6)
        end if
     end do
```

```
      call msg_bug (" Syntax table: unused rules")
    end if
  end subroutine syntax_init_from_ifile
```

For a given rule defined in the input, the first task is to determine its type and key. With these, we can initialize the rule in the table, postponing the association of children.

⟨*Syntax: subroutines*⟩+≡
```
  subroutine set_rule_type_and_key (rule, string, lexer)
    type(syntax_rule_t), intent(inout) :: rule
    type(string_t), intent(in) :: string
    type(lexer_t), intent(inout) :: lexer
    type(stream_t), target :: stream
    type(lexeme_t) :: lexeme
    type(string_t) :: key
    character(2) :: type
    call lexer_clear (lexer)
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    call lex (lexeme, lexer)
    type = lexeme_get_string (lexeme)
    call lex (lexeme, lexer)
    key = lexeme_get_contents (lexeme)
    call stream_final (stream)
    if (trim (key) /= "") then
       select case (type)
       case ("IG");  call syntax_rule_init (rule, key, S_IGNORE)
       case ("LO");  call syntax_rule_init (rule, key, S_LOGICAL)
       case ("IN");  call syntax_rule_init (rule, key, S_INTEGER)
       case ("RE");  call syntax_rule_init (rule, key, S_REAL)
       case ("CO");  call syntax_rule_init (rule, key, S_COMPLEX)
       case ("ID");  call syntax_rule_init (rule, key, S_IDENTIFIER)
       case ("KE");  call syntax_rule_init (rule, key, S_KEYWORD)
       case ("QU");  call syntax_rule_init (rule, key, S_QUOTED)
       case ("SE");  call syntax_rule_init (rule, key, S_SEQUENCE)
       case ("GR");  call syntax_rule_init (rule, key, S_GROUP)
       case ("LI");  call syntax_rule_init (rule, key, S_LIST)
       case ("AR");  call syntax_rule_init (rule, key, S_ARGS)
       case ("AL");  call syntax_rule_init (rule, key, S_ALTERNATIVE)
       case default
          call lexer_show_location (lexer)
          call msg_bug (" Syntax definition: unknown type '" // type // "'")
       end select
    else
       print *, char (string)
       call msg_bug (" Syntax definition: empty rule key")
    end if
  end subroutine set_rule_type_and_key
```

This function returns the number of tokens in an input line.

⟨*Syntax: subroutines*⟩+≡
```
  function get_n_token (string, lexer) result (n)
    integer :: n
```

```
      type(string_t), intent(in) :: string
      type(lexer_t), intent(inout) :: lexer
      type(stream_t), target :: stream
      type(lexeme_t) :: lexeme
      integer :: i
      call lexer_clear (lexer)
      call stream_init (stream, string)
      call lexer_assign_stream (lexer, stream)
      i = 0
      do
         call lex (lexeme, lexer)
         if (lexeme_is_break (lexeme))  exit
         i = i + 1
      end do
      n = i
      call stream_final (stream)
  end function get_n_token
```

This subroutine extracts the rule contents for an input line. There are three tasks: (1) determine the number of children, depending on the rule type; (2) find and set the separator and delimiter strings, if required; (3) scan the child rules, find them in the syntax table and associate the parent rule with them.

⟨*Syntax: subroutines*⟩+≡

```
    subroutine set_rule_contents (rule, syntax, n_token, string, lexer)
      type(syntax_rule_t), intent(inout) :: rule
      type(syntax_t), intent(in), target :: syntax
      integer, intent(in) :: n_token
      type(string_t), intent(in) :: string
      type(lexer_t), intent(inout) :: lexer
      type(stream_t), target :: stream
      type(lexeme_t), dimension(n_token) :: lexeme
      integer :: i, n_children
      call lexer_clear (lexer)
      call stream_init (stream, string)
      call lexer_assign_stream (lexer, stream)
      do i = 1, n_token
         call lex (lexeme(i), lexer)
      end do
      call stream_final (stream)
      n_children = get_n_children ()
      call set_delimiters
      if (n_children > 1)  call set_separator
      if (n_children > 0)  call set_children
    contains
      function get_n_children () result (n_children)
        integer :: n_children
        select case (rule%type)
        case (S_QUOTED)
           if (n_token /= 6)  call broken_rule (rule)
           n_children = 0
        case (S_GROUP)
           if (n_token /= 6)  call broken_rule (rule)
           n_children = 1
```

179

```
      case (S_SEQUENCE)
         if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
            if (n_token <= 4)  call broken_rule (rule)
            call syntax_rule_set_modifier &
                 (rule, lexeme_get_string (lexeme(n_token)))
            n_children = n_token - 4
         else
            if (n_token <= 3) call broken_rule (rule)
            n_children = n_token - 3
         end if
      case (S_LIST)
         if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
            if (n_token <= 4 .or. mod (n_token, 2) /= 1) &
                 call broken_rule (rule)
            call syntax_rule_set_modifier &
                 (rule, lexeme_get_string (lexeme(n_token)))
         else if (n_token <= 3 .or. mod (n_token, 2) /= 0) then
            call broken_rule (rule)
         end if
         n_children = (n_token - 2) / 2
      case (S_ARGS)
         if (is_modifier (lexeme_get_string (lexeme(n_token-1)))) then
            if (n_token <= 6 .or. mod (n_token, 2) /= 1) &
                 call broken_rule (rule)
            call syntax_rule_set_modifier &
                 (rule, lexeme_get_string (lexeme(n_token-1)))
         else if (n_token <= 5 .or. mod (n_token, 2) /= 0) then
            call broken_rule (rule)
         end if
         n_children = (n_token - 4) / 2
      case (S_ALTERNATIVE)
         if (n_token <= 3 .or. mod (n_token, 2) /= 0)  call broken_rule (rule)
         n_children = (n_token - 2) / 2
      end select
   end function get_n_children
   subroutine set_delimiters
     type(string_t), dimension(2) :: delimiter
     select case (rule%type)
     case (S_QUOTED, S_GROUP, S_ARGS)
        delimiter(1) = lexeme_get_contents (lexeme(4))
        delimiter(2) = lexeme_get_contents (lexeme(n_token))
        call syntax_rule_set_delimiter (rule, delimiter)
     end select
   end subroutine set_delimiters
   subroutine set_separator
     type(string_t) :: separator
     select case (rule%type)
     case (S_LIST)
        separator = lexeme_get_contents (lexeme(5))
        call syntax_rule_set_separator (rule, separator)
     case (S_ARGS)
        separator = lexeme_get_contents (lexeme(6))
        call syntax_rule_set_separator (rule, separator)
     end select
```

```
       end subroutine set_separator
    subroutine set_children
      allocate (rule%child(n_children))
      select case (rule%type)
      case (S_GROUP)
         call syntax_rule_set_sub (rule, 1, syntax_get_rule_ptr (syntax, &
              lexeme_get_contents (lexeme(5))))
      case (S_SEQUENCE)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(i+3))))
         end do
      case (S_LIST, S_ALTERNATIVE)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(2*i+2))))
         end do
      case (S_ARGS)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(2*i+3))))
         end do
      end select
    end subroutine set_children
    subroutine broken_rule (rule)
      type(syntax_rule_t), intent(in) :: rule
      call lexer_show_location (lexer)
      call msg_bug (" Syntax definition: broken rule '" &
           // char (wkey (rule)) // "'")
    end subroutine broken_rule
  end subroutine set_rule_contents
```

This routine completes the syntax table object. We assume that the rule array
is set up. We associate the top rule with the first entry in the rule array and
build up the keyword list.

The keyword list includes delimiters and separators. Filling it can only be
done after all rules are set. We scan the rule tree. For each keyword that
we find, we try to add it to the keyword list; the pointer to the last element
is carried along with the recursive scanning. Before appending a keyword, we
check whether it is already in the list.

⟨*Syntax: subroutines*⟩+≡
```
  subroutine syntax_make_keyword_list (syntax)
    type(syntax_t), intent(inout), target :: syntax
    type(syntax_rule_t), pointer :: rule
    rule => syntax%rule(1)
    call rule_scan_rec (rule, syntax%keyword_list)
  contains
    recursive subroutine rule_scan_rec (rule, keyword_list)
      type(syntax_rule_t), pointer :: rule
      type(keyword_list_t), intent(inout) :: keyword_list
      integer :: i
      if (rule%used)  return
      rule%used = .true.
```

```fortran
    select case (rule%type)
    case (S_UNKNOWN)
       call msg_bug (" Syntax: rule tree contains undefined rule")
    case (S_KEYWORD)
       call keyword_list_add (keyword_list, rule%keyword)
    end select
    select case (rule%type)
    case (S_LIST, S_ARGS)
       call keyword_list_add (keyword_list, rule%separator)
    end select
    select case (rule%type)
    case (S_GROUP, S_ARGS)
       call keyword_list_add (keyword_list, rule%delimiter(1))
       call keyword_list_add (keyword_list, rule%delimiter(2))
    end select
    select case (rule%type)
    case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
       if (.not. allocated (rule%child)) &
            call msg_bug (" Syntax: Non-terminal rule without children")
    case default
       if (allocated (rule%child)) &
            call msg_bug (" Syntax: Terminal rule with children")
    end select
    if (allocated (rule%child)) then
       do i = 1, size (rule%child)
          call rule_scan_rec (rule%child(i)%p, keyword_list)
       end do
    end if
  end subroutine rule_scan_rec
end subroutine syntax_make_keyword_list
```

The finalizer deallocates the rule pointer array and deletes the keyword list.

⟨*Syntax: subroutines*⟩+≡
```fortran
subroutine syntax_final (syntax)
  type(syntax_t), intent(inout) :: syntax
  if (allocated (syntax%rule))  deallocate (syntax%rule)
  call keyword_list_final (syntax%keyword_list)
end subroutine syntax_final
```

3.3.6 Accessing the syntax table

Return a pointer to the top rule

⟨*Syntax: public*⟩+≡
```fortran
public :: syntax_get_top_rule_ptr
```

⟨*Syntax: subroutines*⟩+≡
```fortran
function syntax_get_top_rule_ptr (syntax) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(syntax_t), intent(in), target :: syntax
  if (allocated (syntax%rule)) then
     rule => syntax%rule(1)
  else
```

```
                  rule => null ()
             end if
        end function syntax_get_top_rule_ptr
```

Assign the pointer to the rule associated with a given key (assumes that the
rule array is allocated)

⟨*Syntax: public*⟩+≡
```
        public :: syntax_get_rule_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
        function syntax_get_rule_ptr (syntax, key) result (rule)
          type(syntax_rule_t), pointer :: rule
          type(syntax_t), intent(in), target :: syntax
          type(string_t), intent(in) :: key
          integer :: i
          do i = 1, size (syntax%rule)
             if (syntax%rule(i)%keyword == key) then
                rule => syntax%rule(i)
                return
             end if
          end do
          call msg_bug (" Syntax table: Rule " // char (key) // " not found")
        end function syntax_get_rule_ptr
```

Return a pointer to the keyword list

⟨*Syntax: public*⟩+≡
```
        public :: syntax_get_keyword_list_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
        function syntax_get_keyword_list_ptr (syntax) result (keyword_list)
          type(keyword_list_t), pointer :: keyword_list
          type(syntax_t), intent(in), target :: syntax
          keyword_list => syntax%keyword_list
        end function syntax_get_keyword_list_ptr
```

3.3.7 I/O

Write a readable representation of the syntax table

⟨*Syntax: public*⟩+≡
```
        public :: syntax_write
```

⟨*Syntax: subroutines*⟩+≡
```
        subroutine syntax_write (syntax, unit)
          type(syntax_t), intent(in) :: syntax
          integer, intent(in), optional :: unit
          integer :: u
          integer :: i
          u = output_unit (unit);  if (u < 0)  return
          write (u, "(A)") "Syntax table:"
          if (allocated (syntax%rule)) then
             do i = 1, size (syntax%rule)
                call syntax_rule_write (syntax%rule(i), u)
```

```
      end do
  else
      write (u, "(1x,A)") "[not allocated]"
  end if
  call keyword_list_write (syntax%keyword_list, u)
end subroutine syntax_write
```

3.4 The parser

On a small scale, the parser interprets the string tokens returned by the lexer; they are interpreted as numbers, keywords and such and stored as a typed object. On a large scale, a text is read, parsed, and a syntax rule set is applied such that the tokens are stored as a parse tree. Syntax errors are spotted in this process, so the resulting parse tree is syntactically correct by definition.

⟨parser.f90⟩≡
  ⟨*File header*⟩

  ```
  module parser
  ```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ```
    use limits, only: DIGITS, FMT_19 !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use md5
    use unit_tests
    use lexers
    use syntax_rules
  ```

  ⟨*Standard module head*⟩

  ⟨*Parser: public*⟩

  ⟨*Parser: types*⟩

  ⟨*Parser: interfaces*⟩

  ```
  contains
  ```

  ⟨*Parser: procedures*⟩

  ⟨*Parser: tests*⟩

  ```
  end module parser
  ```

3.4.1 The token type

Tokens are elements of the parsed input that carry a value: logical, integer, real, quoted string, (unquoted) identifier, or known keyword. Note that non-keyword tokens also have an abstract key attached to them.

This is an obvious candidate for polymorphism.

⟨*Parser: types*⟩≡
```
    type :: token_t
       private
       integer :: type = S_UNKNOWN
       logical, pointer :: lval => null ()
       integer, pointer :: ival => null ()
       real(default), pointer :: rval => null ()
       complex(default), pointer :: cval => null ()
```

```
      type(string_t), pointer :: sval => null ()
      type(string_t), pointer :: kval => null ()
      type(string_t), dimension(:), pointer :: quote => null ()
   end type token_t
```

Create a token from the lexeme returned by the lexer: Allocate storage and try
to interpret the lexeme according to the type that is requested by the parser.
For a keyword token, match the lexeme against the requested key. If successful,
set the token type, value, and key. Otherwise, set the type to S_UNKNOWN.

⟨*Parser: procedures*⟩≡

```
   subroutine token_init (token, lexeme, requested_type, key)
     type(token_t), intent(out) :: token
     type(lexeme_t), intent(in) :: lexeme
     integer, intent(in) :: requested_type
     type(string_t), intent(in) :: key
     integer :: type
     type = lexeme_get_type (lexeme)
     token%type = S_UNKNOWN
     select case (requested_type)
     case (S_LOGICAL)
        if (type == T_IDENTIFIER)  call read_logical &
            (char (lexeme_get_string (lexeme)))
     case (S_INTEGER)
        if (type == T_NUMERIC)  call read_integer &
            (char (lexeme_get_string (lexeme)))
     case (S_REAL)
        if (type == T_NUMERIC)  call read_real &
            (char (lexeme_get_string (lexeme)))
     case (S_COMPLEX)
        if (type == T_NUMERIC)  call read_complex &
            (char (lexeme_get_string (lexeme)))
     case (S_IDENTIFIER)
        if (type == T_IDENTIFIER)  call read_identifier &
            (lexeme_get_string (lexeme))
     case (S_KEYWORD)
        if (type == T_KEYWORD)  call check_keyword &
            (lexeme_get_string (lexeme), key)
     case (S_QUOTED)
        if (type == T_QUOTED)   call read_quoted &
            (lexeme_get_contents (lexeme), lexeme_get_delimiters (lexeme))
     case default
        print *, requested_type
        call msg_bug (" Invalid token type code requested by the parser")
     end select
     if (token%type /= S_UNKNOWN) then
        allocate (token%kval)
        token%kval = key
     end if
   contains
     subroutine read_logical (s)
       character(*), intent(in) :: s
       select case (s)
       case ("t", "T", "true", "TRUE", "y", "Y", "yes", "YES")
```

```fortran
      allocate (token%lval)
      token%lval = .true.
      token%type = S_LOGICAL
  case ("f", "F", "false", "FALSE", "n", "N", "no", "NO")
      allocate (token%lval)
      token%lval = .false.
      token%type = S_LOGICAL
  end select
end subroutine read_logical
subroutine read_integer (s)
  character(*), intent(in) :: s
  integer :: tmp, iostat
  if (verify (s, DIGITS) == 0) then
      read (s, *, iostat=iostat) tmp
      if (iostat == 0) then
         allocate (token%ival)
         token%ival = tmp
         token%type = S_INTEGER
      end if
  end if
end subroutine read_integer
subroutine read_real (s)
  character(*), intent(in) :: s
  real(default) :: tmp
  integer :: iostat
  read (s, *, iostat=iostat) tmp
  if (iostat == 0) then
      allocate (token%rval)
      token%rval = tmp
      token%type = S_REAL
  end if
end subroutine read_real
subroutine read_complex (s)
  character(*), intent(in) :: s
  complex(default) :: tmp
  integer :: iostat
  read (s, *, iostat=iostat) tmp
  if (iostat == 0) then
      allocate (token%cval)
      token%cval = tmp
      token%type = S_COMPLEX
  end if
end subroutine read_complex
subroutine read_identifier (s)
  type(string_t), intent(in) :: s
  allocate (token%sval)
  token%sval = s
  token%type = S_IDENTIFIER
end subroutine read_identifier
subroutine check_keyword (s, key)
  type(string_t), intent(in) :: s
  type(string_t), intent(in) :: key
  if (key == s)  token%type = S_KEYWORD
end subroutine check_keyword
```

```
      subroutine read_quoted (s, del)
        type(string_t), intent(in) :: s
        type(string_t), dimension(2), intent(in) :: del
        allocate (token%sval, token%quote(2))
        token%sval = s
        token%quote(1) = del(1)
        token%quote(2) = del(2)
        token%type = S_QUOTED
      end subroutine read_quoted
    end subroutine token_init
```

Manually set a token to a keyword.

⟨*Parser: procedures*⟩+≡
```
    subroutine token_init_key (token, key)
      type(token_t), intent(out) :: token
      type(string_t), intent(in) :: key
      token%type = S_KEYWORD
      allocate (token%kval)
      token%kval = key
    end subroutine token_init_key
```

Reset a token to an empty state, freeing allocated memory, and deallocate the
token itself.

⟨*Parser: procedures*⟩+≡
```
    subroutine token_final (token)
      type(token_t), intent(inout) :: token
      token%type = S_UNKNOWN
      if (associated (token%lval))  deallocate (token%lval)
      if (associated (token%ival))  deallocate (token%ival)
      if (associated (token%rval))  deallocate (token%rval)
      if (associated (token%sval))  deallocate (token%sval)
      if (associated (token%kval))  deallocate (token%kval)
      if (associated (token%quote))  deallocate (token%quote)
    end subroutine token_final
```

Check for empty=valid token:

⟨*Parser: procedures*⟩+≡
```
    function token_is_valid (token) result (valid)
      logical :: valid
      type(token_t), intent(in) :: token
      valid = token%type /= S_UNKNOWN
    end function token_is_valid
```

Write the contents of a token.

⟨*Parser: procedures*⟩+≡
```
    subroutine token_write (token, unit)
      type(token_t), intent(in) :: token
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (token%type)
      case (S_LOGICAL)
```

```
      write (u, "(L1)") token%lval
   case (S_INTEGER)
      write (u, "(I0)") token%ival
   case (S_REAL)
      write (u, "(" // FMT_19 // ")") token%rval
   case (S_COMPLEX)
      write (u, "('(',"  // FMT_19 // "',','," // FMT_19 // "',')')") token%cval
   case (S_IDENTIFIER)
      write (u, "(A)") char (token%sval)
   case (S_KEYWORD)
      write (u, "(A,A)") '[keyword] ' // char (token%kval)
   case (S_QUOTED)
      write (u, "(A)") &
            char (token%quote(1)) // char (token%sval) // char (token%quote(2))
   case default
      write (u, "(A)") '[empty]'
   end select
end subroutine token_write
```

Token assignment via deep copy. This is useful to avoid confusion when the
token is transferred to some parse-tree node.

⟨*Parser: interfaces*⟩≡
```
   interface assignment(=)
      module procedure token_assign
      module procedure token_assign_integer
      module procedure token_assign_real
      module procedure token_assign_complex
      module procedure token_assign_logical
      module procedure token_assign_string
   end interface
```

We need to copy only the contents that are actually assigned, the other pointers
remain disassociated.

⟨*Parser: procedures*⟩+≡
```
   subroutine token_assign (token, token_in)
     type(token_t), intent(out) :: token
     type(token_t), intent(in) :: token_in
     token%type = token_in%type
     select case (token%type)
     case (S_LOGICAL);    allocate (token%lval);  token%lval = token_in%lval
     case (S_INTEGER);    allocate (token%ival);  token%ival = token_in%ival
     case (S_REAL);       allocate (token%rval);  token%rval = token_in%rval
     case (S_COMPLEX); allocate (token%cval);  token%cval = token_in%cval
     case (S_IDENTIFIER); allocate (token%sval);  token%sval = token_in%sval
     case (S_QUOTED);     allocate (token%sval);  token%sval = token_in%sval
        allocate (token%quote(2));  token%quote = token_in%quote
     end select
     if (token%type /= S_UNKNOWN) then
        allocate (token%kval);  token%kval = token_in%kval
     end if
   end subroutine token_assign
```

We need to copy only the contents that are actually assigned, the other pointers
remain disassociated.

⟨*Parser: procedures*⟩+≡

```
subroutine token_assign_integer (token, ival)
  type(token_t), intent(out) :: token
  integer, intent(in) :: ival
  token%type = S_INTEGER
  allocate (token%ival)
  token%ival = ival
end subroutine token_assign_integer

subroutine token_assign_real (token, rval)
  type(token_t), intent(out) :: token
  real(default), intent(in) :: rval
  token%type = S_REAL
  allocate (token%rval)
  token%rval = rval
end subroutine token_assign_real

subroutine token_assign_complex (token, cval)
  type(token_t), intent(out) :: token
  complex(default), intent(in) :: cval
  token%type = S_COMPLEX
  allocate (token%cval)
  token%cval = cval
end subroutine token_assign_complex

subroutine token_assign_logical (token, lval)
  type(token_t), intent(out) :: token
  logical, intent(in) :: lval
  token%type = S_LOGICAL
  allocate (token%lval)
  token%lval = lval
end subroutine token_assign_logical

subroutine token_assign_string (token, sval)
  type(token_t), intent(out) :: token
  type(string_t), intent(in) :: sval
  token%type = S_QUOTED
  allocate (token%sval)
  token%sval = sval
  allocate (token%quote(2));  token%quote = '"'
end subroutine token_assign_string
```

3.4.2   Retrieve token contents

These functions all do a trivial sanity check that should avoid crashes.

⟨*Parser: procedures*⟩+≡

```
function token_get_logical (token) result (lval)
  logical :: lval
  type(token_t), intent(in) :: token
  if (associated (token%lval)) then
```

```
      lval = token%lval
   else
      call token_mismatch (token, "logical")
   end if
end function token_get_logical

function token_get_integer (token) result (ival)
   integer :: ival
   type(token_t), intent(in) :: token
   if (associated (token%ival)) then
      ival = token%ival
   else
      call token_mismatch (token, "integer")
   end if
end function token_get_integer

function token_get_real (token) result (rval)
   real(default) :: rval
   type(token_t), intent(in) :: token
   if (associated (token%rval)) then
      rval = token%rval
   else
      call token_mismatch (token, "real")
   end if
end function token_get_real

function token_get_cmplx (token) result (cval)
   complex(default) :: cval
   type(token_t), intent(in) :: token
   if (associated (token%cval)) then
      cval = token%cval
   else
      call token_mismatch (token, "complex")
   end if
end function token_get_cmplx

function token_get_string (token) result (sval)
   type(string_t) :: sval
   type(token_t), intent(in) :: token
   if (associated (token%sval)) then
      sval = token%sval
   else
      call token_mismatch (token, "string")
   end if
end function token_get_string

function token_get_key (token) result (kval)
   type(string_t) :: kval
   type(token_t), intent(in) :: token
   if (associated (token%kval)) then
      kval = token%kval
   else
      call token_mismatch (token, "keyword")
   end if
```

```
    end function token_get_key

    function token_get_quote (token) result (quote)
      type(string_t), dimension(2) :: quote
      type(token_t), intent(in) :: token
      if (associated (token%quote)) then
         quote = token%quote
      else
         call token_mismatch (token, "quote")
      end if
    end function token_get_quote
```

⟨*Parser: procedures*⟩+≡
```
    subroutine token_mismatch (token, type)
      type(token_t), intent(in) :: token
      character(*), intent(in) :: type
      write (6, "(A)", advance="no")  "Token: "
      call token_write (token)
      call msg_bug (" Token type mismatch; value required as " // type)
    end subroutine token_mismatch
```

3.4.3   The parse tree: nodes

The parser will generate a parse tree from the input stream. Each node in this
parse tree points to the syntax rule that was applied. (Since syntax rules are
stored in a pointer-type array within the syntax table, they qualify as targets.)
A leaf node contains a token. A branch node has subnodes. The subnodes are
stored as a list, so each node also has a **next** pointer.

⟨*Parser: public*⟩≡
```
    public :: parse_node_t
```

⟨*Parser: types*⟩+≡
```
    type :: parse_node_t
      private
      type(syntax_rule_t), pointer :: rule => null ()
      type(token_t) :: token
      integer :: n_sub = 0
      type(parse_node_t), pointer :: sub_first => null ()
      type(parse_node_t), pointer :: sub_last => null ()
      type(parse_node_t), pointer :: next => null ()
     contains
    ⟨Parser: parse node: TBP⟩
    end type parse_node_t
```

Container for parse node pointers, useful for creating pointer arrays:

⟨*Parser: public*⟩+≡
```
    public :: parse_node_p
```

⟨*Parser: types*⟩+≡
```
    type :: parse_node_p
      type(parse_node_t), pointer :: ptr => null ()
    end type parse_node_p
```

Output. The first version writes a node together with its sub-node tree, organized by indentation.

⟨*Parser: parse node: TBP*⟩≡
```
    procedure :: write => parse_node_write_rec
```

⟨*Parser: public*⟩+≡
```
    public :: parse_node_write_rec
```

⟨*Parser: procedures*⟩+≡
```
    recursive subroutine parse_node_write_rec (node, unit, short, depth)
      class(parse_node_t), intent(in), target :: node
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: short
      integer, intent(in), optional :: depth
      integer :: u, d
      type(parse_node_t), pointer :: current
      u = output_unit (unit);  if (u < 0)  return
      d = 0;  if (present (depth))  d = depth
      call parse_node_write (node, u, short=short)
      current => node%sub_first
      do while (associated (current))
         write (u, "(A)", advance = "no")  repeat ("|  ", d)
         call parse_node_write_rec (current, unit, short, d+1)
         current => current%next
      end do
    end subroutine parse_node_write_rec
```

This does the actual output for a single node, without recursion.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_write
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_write (node, unit, short)
      class(parse_node_t), intent(in) :: node
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: short
      integer :: u
      type(parse_node_t), pointer :: current
      u = output_unit (unit);  if (u < 0)  return
      write (u, "('+ ')", advance = "no")
      if (associated (node%rule)) then
         call syntax_rule_write (node%rule, u, &
              short=short, key_only=.true., advance=.false.)
         if (token_is_valid (node%token)) then
            write (u, "('  = ')", advance="no")
            call token_write (node%token, u)
         else if (associated (node%sub_first)) then
            write (u, "('  = ')", advance="no")
            current => node%sub_first
            do while (associated (current))
               call syntax_rule_write (current%rule, u, &
                    short=.true., key_only=.true., advance=.false.)
               current => current%next
            end do
            write (u, *)
```

193

```
         else
             write (u, *)
         end if
      else
         write (u, *) "[empty]"
      end if
   end subroutine parse_node_write
```

Finalize the token and recursively finalize and deallocate all sub-nodes.

⟨*Parser: public*⟩+≡
```
   public :: parse_node_final
```

⟨*Parser: procedures*⟩+≡
```
   recursive subroutine parse_node_final (node, recursive)
     type(parse_node_t), intent(inout) :: node
     type(parse_node_t), pointer :: current
     logical, intent(in), optional :: recursive
     logical :: rec
     rec = .true.;  if (present (recursive))  rec = recursive
     call token_final (node%token)
     if (rec) then
        do while (associated (node%sub_first))
           current => node%sub_first
           node%sub_first => node%sub_first%next
           call parse_node_final (current)
           deallocate (current)
        end do
     end if
   end subroutine parse_node_final
```

3.4.4   Filling nodes

The constructors allocate and initialize the node. There are two possible initializers (in a later version, should correspond to different type extensions).

First, leaf (terminal) nodes. The token constructor does the actual work, looking at the requested type and key for the given rule and matching against the lexeme contents. If it fails, the token will keep the type S_UNKNOWN and remain empty. Otherwise, we have a valid node which contains the new token.

If the lexeme argument is absent, the token is left empty.

⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_create_leaf (node, rule, lexeme)
     type(parse_node_t), pointer :: node
     type(syntax_rule_t), intent(in), target :: rule
     type(lexeme_t), intent(in) :: lexeme
     allocate (node)
     node%rule => rule
     call token_init (node%token, lexeme, &
          syntax_rule_get_type (rule), syntax_rule_get_key (rule))
     if (.not. token_is_valid (node%token))  deallocate (node)
   end subroutine parse_node_create_leaf
```

This version allows us to manually create a leaf node that holds a keyword.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_create_key
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_create_key (node, rule)
      type(parse_node_t), intent(out) :: node
      type(syntax_rule_t), intent(in), target :: rule
      node%rule => rule
      call token_init_key (node%token, syntax_rule_get_key (rule))
    end subroutine parse_node_create_key
```

This version allows us to manually create a leaf node that holds a fixed value.
Only one of the optional values should be provided.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_create_value
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_create_value (node, rule, ival, rval, cval, sval, lval)
      type(parse_node_t), intent(out) :: node
      type(syntax_rule_t), intent(in), target :: rule
      integer, intent(in), optional :: ival
      real(default), intent(in), optional :: rval
      complex(default), intent(in), optional :: cval
      type(string_t), intent(in), optional :: sval
      logical, intent(in), optional :: lval
      node%rule => rule
      call parse_node_set_value (node, ival, rval, cval, sval, lval)
    end subroutine parse_node_create_value
```

Directly set the value without changing anything else.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_set_value
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_set_value (node, ival, rval, cval, sval, lval)
      type(parse_node_t), intent(inout) :: node
      integer, intent(in), optional :: ival
      real(default), intent(in), optional :: rval
      complex(default), intent(in), optional :: cval
      type(string_t), intent(in), optional :: sval
      logical, intent(in), optional :: lval
      if (present (ival)) then
         node%token = ival
      else if (present (rval)) then
         node%token = rval
      else if (present (cval)) then
         node%token = cval
      else if (present (lval)) then
         node%token = lval
      else if (present (sval)) then
         node%token = sval
      end if
    end subroutine parse_node_set_value
```

Second, branch nodes. We first assign the rule:

⟨Parser: public⟩+≡
```
    public :: parse_node_create_branch
```

⟨Parser: procedures⟩+≡
```
    subroutine parse_node_create_branch (node, rule)
      type(parse_node_t), pointer :: node
      type(syntax_rule_t), intent(in), target :: rule
      allocate (node)
      node%rule => rule
    end subroutine parse_node_create_branch
```

Append a sub-node. The sub-node must exist and be a valid target, otherwise nothing is done.

⟨Parser: public⟩+≡
```
    public :: parse_node_append_sub
```

⟨Parser: procedures⟩+≡
```
    subroutine parse_node_append_sub (node, sub)
      type(parse_node_t), intent(inout) :: node
      type(parse_node_t), pointer :: sub
      if (associated (sub)) then
         if (associated (node%sub_last)) then
            node%sub_last%next => sub
         else
            node%sub_first => sub
         end if
         node%sub_last => sub
      end if
    end subroutine parse_node_append_sub
```

For easy access, once the list is complete we count the number of sub-nodes. If there are no subnodes, the whole node is deleted.

⟨Parser: public⟩+≡
```
    public :: parse_node_freeze_branch
```

⟨Parser: procedures⟩+≡
```
    subroutine parse_node_freeze_branch (node)
      type(parse_node_t), pointer :: node
      type(parse_node_t), pointer :: current
      node%n_sub = 0
      current => node%sub_first
      do while (associated (current))
         node%n_sub = node%n_sub + 1
         current => current%next
      end do
      if (node%n_sub == 0)  deallocate (node)
    end subroutine parse_node_freeze_branch
```

Replace the syntax rule. This makes sense only if the parse node adheres to the syntax of the new rule.

⟨Parser: public⟩+≡
```
    public :: parse_node_replace_rule
```

⟨Parser: procedures⟩+≡
```
    subroutine parse_node_replace_rule (node, rule)
      type(parse_node_t), pointer :: node
      type(syntax_rule_t), intent(in), target :: rule
      node%rule => rule
    end subroutine parse_node_replace_rule
```

Replace the last subnode by the target node. Since the subnodes are stored as a linked list, we can do this only if we copy the subnodes. Furthermore, the target node must also be copied, and the `next` pointer of the copy is nullified. This ensures that we cannot modify the originals at the subnode level.

All copies are shallow copies. This implies that further modifications at the sub-subnode level will affect the original nodes and must therefore be forbidden.

Use with care, this invites to memory mismanagement. The copy nodes can be deallocated, but not finalized, since its subnodes are the same objects as the subnodes of the target node.

⟨Parser: public⟩+≡
```
    public :: parse_node_replace_last_sub
```

⟨Parser: procedures⟩+≡
```
    subroutine parse_node_replace_last_sub (node, pn_target)
      type(parse_node_t), intent(inout), target :: node
      type(parse_node_t), intent(in), target :: pn_target
      type(parse_node_t), pointer :: current, current_copy, previous
      integer :: i
      select case (node%n_sub)
      case (1)
         allocate (current_copy)
         current_copy = pn_target
         node%sub_first => current_copy
      case (2:)
         current => node%sub_first
         allocate (current_copy)
         current_copy = current
         node%sub_first => current_copy
         previous => current_copy
         do i = 1, node%n_sub - 2
            current => current%next
            allocate (current_copy)
            current_copy = current
            previous%next => current_copy
            previous => current_copy
         end do
         allocate (current_copy)
         current_copy = pn_target
         previous%next => current_copy
      case default
         call parse_node_write (node)
         call msg_bug ("'replace_last_sub' called for non-branch parse node")
      end select
      current_copy%next => null ()
      node%sub_last => current_copy
    end subroutine parse_node_replace_last_sub
```

3.4.5 Accessing nodes

Return the node contents. Check if pointers are associated. No check when accessing a sub-node; assume that `parse_node_n_sub` is always used for the upper bound.

The token extractor returns a pointer.

⟨*Parser: public*⟩+≡
```
public :: parse_node_get_rule_ptr
public :: parse_node_get_n_sub
public :: parse_node_get_sub_ptr
public :: parse_node_get_next_ptr
public :: parse_node_get_last_sub_ptr
```

⟨*Parser: procedures*⟩+≡
```
function parse_node_get_rule_ptr (node) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(parse_node_t), intent(in), target :: node
  if (associated (node%rule)) then
     rule => node%rule
  else
     rule => null ()
     call parse_node_undefined (node, "rule")
  end if
end function parse_node_get_rule_ptr

function parse_node_get_n_sub (node) result (n)
  integer :: n
  type(parse_node_t), intent(in) :: node
  n = node%n_sub
end function parse_node_get_n_sub

function parse_node_get_sub_ptr (node, n, tag, required) result (sub)
  type(parse_node_t), pointer :: sub
  type(parse_node_t), intent(in), target :: node
  integer, intent(in), optional :: n
  character(*), intent(in), optional :: tag
  logical, intent(in), optional :: required
  integer :: i
  sub => node%sub_first
  if (present (n)) then
     do i = 2, n
        if (associated (sub)) then
           sub => sub%next
        else
           return
        end if
     end do
  end if
  call parse_node_check (sub, tag, required)
end function parse_node_get_sub_ptr

function parse_node_get_next_ptr (sub, n, tag, required) result (next)
```

```
      type(parse_node_t), pointer :: next
      type(parse_node_t), intent(in), target :: sub
      integer, intent(in), optional :: n
      character(*), intent(in), optional :: tag
      logical, intent(in), optional :: required
      integer :: i
      next => sub%next
      if (present (n)) then
         do i = 2, n
            if (associated (next)) then
               next => next%next
            else
               exit
            end if
         end do
      end if
      call parse_node_check (next, tag, required)
   end function parse_node_get_next_ptr

   function parse_node_get_last_sub_ptr (node, tag, required) result (sub)
     type(parse_node_t), pointer :: sub
     type(parse_node_t), intent(in), target :: node
     character(*), intent(in), optional :: tag
     logical, intent(in), optional :: required
     sub => node%sub_last
     call parse_node_check (sub, tag, required)
   end function parse_node_get_last_sub_ptr
```

⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_undefined (node, obj)
     type(parse_node_t), intent(in) :: node
     character(*), intent(in) :: obj
     call parse_node_write (node, 6)
     call msg_bug (" Parse-tree node: " // obj // " requested, but undefined")
   end subroutine parse_node_undefined
```

Check if a parse node has a particular tag, and if it is associated:
⟨*Parser: public*⟩+≡
```
   public :: parse_node_check
```
⟨*Parser: procedures*⟩+≡
```
   subroutine parse_node_check (node, tag, required)
     type(parse_node_t), pointer :: node
     character(*), intent(in), optional :: tag
     logical, intent(in), optional :: required
     if (associated (node)) then
        if (present (tag)) then
           if (parse_node_get_rule_key (node) /= tag) &
                call parse_node_mismatch (tag, node)
        end if
     else
        if (present (required)) then
           if (required) &
                call msg_bug (" Missing node, expected <" // tag // ">")
```

```
        end if
      end if
    end subroutine parse_node_check
```

This is called by a parse-tree scanner if the expected and the actual nodes do not match

⟨*Parser: public*⟩+≡
```
    public :: parse_node_mismatch
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_node_mismatch (string, parse_node)
      character(*), intent(in) :: string
      type(parse_node_t), intent(in) :: parse_node
      call parse_node_write (parse_node)
      call msg_bug (" Syntax mismatch, expected <" // string // ">.")
    end subroutine parse_node_mismatch
```

The following functions are wrappers for extracting the token contents.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_get_logical
    public :: parse_node_get_integer
    public :: parse_node_get_real
    public :: parse_node_get_cmplx
    public :: parse_node_get_string
    public :: parse_node_get_key
    public :: parse_node_get_rule_key
```

⟨*Parser: procedures*⟩+≡
```
    function parse_node_get_logical (node) result (lval)
      logical :: lval
      type(parse_node_t), intent(in), target :: node
      lval = token_get_logical (parse_node_get_token_ptr (node))
    end function parse_node_get_logical

    function parse_node_get_integer (node) result (ival)
      integer :: ival
      type(parse_node_t), intent(in), target :: node
      ival = token_get_integer (parse_node_get_token_ptr (node))
    end function parse_node_get_integer

    function parse_node_get_real (node) result (rval)
      real(default) :: rval
      type(parse_node_t), intent(in), target :: node
      rval = token_get_real (parse_node_get_token_ptr (node))
    end function parse_node_get_real

    function parse_node_get_cmplx (node) result (cval)
      complex(default) :: cval
      type(parse_node_t), intent(in), target :: node
      cval = token_get_cmplx (parse_node_get_token_ptr (node))
    end function parse_node_get_cmplx

    function parse_node_get_string (node) result (sval)
      type(string_t) :: sval
```

```
    type(parse_node_t), intent(in), target :: node
    sval = token_get_string (parse_node_get_token_ptr (node))
  end function parse_node_get_string

  function parse_node_get_key (node) result (kval)
    type(string_t) :: kval
    type(parse_node_t), intent(in), target :: node
    kval = token_get_key (parse_node_get_token_ptr (node))
  end function parse_node_get_key

  function parse_node_get_rule_key (node) result (kval)
    type(string_t) :: kval
    type(parse_node_t), intent(in), target :: node
    kval = syntax_rule_get_key (parse_node_get_rule_ptr (node))
  end function parse_node_get_rule_key

  function parse_node_get_token_ptr (node) result (token)
    type(token_t), pointer :: token
    type(parse_node_t), intent(in), target :: node
    if (token_is_valid (node%token)) then
       token => node%token
    else
       call parse_node_undefined (node, "token")
    end if
  end function parse_node_get_token_ptr
```

Return a MD5 sum for a parse node. The method is to write the node to a scratch file and to evaluate the MD5 sum of that file.

⟨*Parser: public*⟩+≡
```
    public :: parse_node_get_md5sum
```

⟨*Parser: procedures*⟩+≡
```
    function parse_node_get_md5sum (pn) result (md5sum_pn)
      character(32) :: md5sum_pn
      type(parse_node_t), intent(in) :: pn
      integer :: u
      u = free_unit ()
      open (unit = u, status = "scratch", action = "readwrite")
      call parse_node_write_rec (pn, unit=u)
      rewind (u)
      md5sum_pn = md5sum (u)
      close (u)
    end function parse_node_get_md5sum
```

3.4.6   The parse tree

The parse tree is a tree of nodes, where leaf nodes hold a valid token, while branch nodes point to a list of sub-nodes.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_t
```

⟨*Parser: types*⟩+≡
```
    type :: parse_tree_t
       private
       type(parse_node_t), pointer :: root_node => null ()
    end type parse_tree_t
```

The parser. Its arguments are the parse tree (which should be empty initially), the lexer (which should be already set up), the syntax table (which should be valid), and the input stream. The input stream is completely parsed, using the lexer setup and the syntax rules as given, and the parse tree is built accordingly.

If `check_eof` is absent or true, the parser will complain about trailing garbage. Otherwise, it will ignore it.

By default, the input stream is matched against the top rule in the specified syntax. If `key` is given, it is matched against the rule with the specified key instead.

Failure at the top level means that no rule could match at all; in this case the error message will show the top rule.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_init
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_init &
         (parse_tree, syntax, lexer, key, check_eof)
      type(parse_tree_t), intent(inout) :: parse_tree
      type(lexer_t), intent(inout) :: lexer
      type(syntax_t), intent(in), target :: syntax
      type(string_t), intent(in), optional :: key
      logical, intent(in), optional :: check_eof
      type(syntax_rule_t), pointer :: rule
      type(lexeme_t) :: lexeme
      type(parse_node_t), pointer :: node
      logical :: ok, check
      check = .true.;  if (present (check_eof)) check = check_eof
      call lexer_clear (lexer)
      if (present (key)) then
         rule => syntax_get_rule_ptr (syntax, key)
      else
         rule => syntax_get_top_rule_ptr (syntax)
      end if
      if (associated (rule)) then
         call parse_node_match_rule (node, rule, ok)
         if (ok) then
            parse_tree%root_node => node
         else
            call parse_error (rule, lexeme)
         end if
         if (check) then
            call lex (lexeme, lexer)
            if (.not. lexeme_is_eof (lexeme)) then
               call lexer_show_location (lexer)
               call msg_fatal (" Syntax error " &
                    // "(at or before the location indicated above)")
            end if
```

```
          end if
       else
          call msg_bug (" Parser failed because syntax is empty")
       end if
    contains
    〈Parser: internal subroutines of parse_tree_init〉
    end subroutine parse_tree_init
```

The parser works recursively, following the rule tree, building the tree of nodes
on the fly. If the given rule matches, the node is filled on return. If not, the
node remains empty.

〈Parser: internal subroutines of parse_tree_init〉≡

```
    recursive subroutine parse_node_match_rule (node, rule, ok)
      type(parse_node_t), pointer :: node
      type(syntax_rule_t), intent(in), target :: rule
      logical, intent(out) :: ok
      logical, parameter :: debug = .false.
      integer :: type
      if (debug)  write (6, "(A)", advance="no") "Parsing rule: "
      if (debug)  call syntax_rule_write (rule, 6)
      node => null ()
      type = syntax_rule_get_type (rule)
      if (syntax_rule_is_atomic (rule)) then
         call lex (lexeme, lexer)
         if (debug)  write (6, "(A)", advance="no") "Token: "
         if (debug)  call lexeme_write (lexeme, 6)
         call parse_node_create_leaf (node, rule, lexeme)
         ok = associated (node)
         if (.not. ok)  call lexer_put_back (lexer, lexeme)
      else
         select case (type)
         case (S_ALTERNATIVE);  call parse_alternative (node, rule, ok)
         case (S_GROUP);        call parse_group (node, rule, ok)
         case (S_SEQUENCE);     call parse_sequence (node, rule, .false., ok)
         case (S_LIST);         call parse_sequence (node, rule, .true., ok)
         case (S_ARGS);         call parse_args (node, rule, ok)
         case (S_IGNORE);       call parse_ignore (node, ok)
         end select
      end if
      if (debug) then
         if (ok) then
            write (6, "(A)", advance="no") "Matched rule: "
         else
            write (6, "(A)", advance="no") "Failed rule: "
         end if
         call syntax_rule_write (rule)
         if (associated (node)) call parse_node_write (node)
      end if
    end subroutine parse_node_match_rule
```

Parse an alternative: try each case. If the match succeeds, the node has been
filled, so return. If nothing works, return failure.

〈Parser: internal subroutines of parse_tree_init〉+≡

```
    recursive subroutine parse_alternative (node, rule, ok)
```

```
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    integer :: i
    do i = 1, syntax_rule_get_n_sub (rule)
       call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, i), ok)
       if (ok)  return
    end do
    ok = .false.
  end subroutine parse_alternative
```

Parse a group: the first and third lexemes have to be the delimiters, the second one is parsed as the actual node, using now the child rule. If the first match fails, return with failure. If the other matches fail, issue an error, since we cannot lex back more than one item.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
  recursive subroutine parse_group (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    type(string_t), dimension(2) :: delimiter
    delimiter = syntax_rule_get_delimiter (rule)
    call lex (lexeme, lexer)
    if (lexeme_get_string (lexeme) == delimiter(1)) then
       call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, 1), ok)
       if (ok) then
          call lex (lexeme, lexer)
          if (lexeme_get_string (lexeme) == delimiter(2)) then
             ok = .true.
          else
             call parse_error (rule, lexeme)
          end if
       else
          call parse_error (rule, lexeme)
       end if
    else
       call lexer_put_back (lexer, lexeme)
       ok = .false.
    end if
  end subroutine parse_group
```

Parsing a sequence. The last rule element may be special: optional and/or repetitive. Each sub-node that matches is appended to the sub-node list of the parent node.

If `sep` is true, we look for a separator after each element.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
  recursive subroutine parse_sequence (node, rule, sep, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(in) :: sep
    logical, intent(out) :: ok
    type(parse_node_t), pointer :: current
    integer :: i, n
    logical :: opt, rep, cont
```

```
        type(string_t) :: separator
        call parse_node_create_branch (node, rule)
        if (sep)  separator = syntax_rule_get_separator (rule)
        n = syntax_rule_get_n_sub (rule)
        opt = syntax_rule_last_optional (rule)
        rep = syntax_rule_last_repetitive (rule)
        ok = .true.
        cont = .true.
        SCAN_RULE: do i = 1, n
           call parse_node_match_rule &
                (current, syntax_rule_get_sub_ptr (rule, i), cont)
           if (cont) then
              call parse_node_append_sub (node, current)
              if (sep .and. (i<n .or. rep)) then
                 call lex (lexeme, lexer)
                 if (lexeme_get_string (lexeme) /= separator) then
                    call lexer_put_back (lexer, lexeme)
                    cont = .false.
                    exit SCAN_RULE
                 end if
              end if
           else
              if (i == n .and. opt) then
                 exit SCAN_RULE
              else if (i == 1) then
                 ok = .false.
                 exit SCAN_RULE
              else
                 call parse_error (rule, lexeme)
              end if
           end if
        end do SCAN_RULE
        if (rep) then
           do while (cont)
              call parse_node_match_rule &
                   (current, syntax_rule_get_sub_ptr (rule, n), cont)
              if (cont) then
                 call parse_node_append_sub (node, current)
                 if (sep) then
                    call lex (lexeme, lexer)
                    if (lexeme_get_string (lexeme) /= separator) then
                       call lexer_put_back (lexer, lexeme)
                       cont = .false.
                    end if
                 end if
              else
                 if (sep)  call parse_error (rule, lexeme)
              end if
           end do
        end if
        call parse_node_freeze_branch (node)
      end subroutine parse_sequence
```

Argument list: We use the `parse_group` code, but call `parse_sequence` inside.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
recursive subroutine parse_args (node, rule, ok)
  type(parse_node_t), pointer :: node
  type(syntax_rule_t), intent(in), target :: rule
  logical, intent(out) :: ok
  type(string_t), dimension(2) :: delimiter
  delimiter = syntax_rule_get_delimiter (rule)
  call lex (lexeme, lexer)
  if (lexeme_get_string (lexeme) == delimiter(1)) then
     call parse_sequence (node, rule, .true., ok)
     if (ok) then
        call lex (lexeme, lexer)
        if (lexeme_get_string (lexeme) == delimiter(2)) then
           ok = .true.
        else
           call parse_error (rule, lexeme)
        end if
     else
        call parse_error (rule, lexeme)
     end if
  else
     call lexer_put_back (lexer, lexeme)
     ok = .false.
  end if
end subroutine parse_args
```

The IGNORE syntax reads one lexeme and discards it if it is numeric, logical or string/identifier (but not a keyword). This is a successful match. Otherwise, the match fails. The node pointer is returned disassociated in any case.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
subroutine parse_ignore (node, ok)
  type(parse_node_t), pointer :: node
  logical, intent(out) :: ok
  call lex (lexeme, lexer)
  select case (lexeme_get_type (lexeme))
  case (T_NUMERIC, T_IDENTIFIER, T_QUOTED)
     ok = .true.
  case default
     ok = .false.
  end select
  node => null ()
end subroutine parse_ignore
```

If the match fails and we cannot step back:

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
subroutine parse_error (rule, lexeme)
  type(syntax_rule_t), intent(in) :: rule
  type(lexeme_t), intent(in) :: lexeme
  character(80) :: buffer
  integer :: u, iostat
  call lexer_show_location (lexer)
  u = free_unit ()
  open (u, status = "scratch")
  write (u, "(A)", advance="no")  "Expected syntax:"
  call syntax_rule_write (rule, u)
  write (u, "(A)", advance="no")  "Found token:"
```

```
    call lexeme_write (lexeme, u)
    rewind (u)
    do
        read (u, "(A)", iostat=iostat)  buffer
        if (iostat /= 0)  exit
        call msg_message (trim (buffer))
    end do
    call msg_fatal (" Syntax error " & 
            // "(at or before the location indicated above)")
end subroutine parse_error
```

The finalizer recursively deallocates all nodes and their contents. For each node,
`parse_node_final` is called on the sub-nodes, which in turn deallocates the
token or sub-node array contained within each of them. At the end, only the
top node remains to be deallocated.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_final
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_final (parse_tree)
      type(parse_tree_t), intent(inout) :: parse_tree
      if (associated (parse_tree%root_node)) then
        call parse_node_final (parse_tree%root_node)
        deallocate (parse_tree%root_node)
      end if
    end subroutine parse_tree_final
```

Print the parse tree. Print one token per line, indented according to the depth
of the node.

The `verbose` version includes type identifiers for the nodes.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_write
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_tree_write (parse_tree, unit, verbose)
      type(parse_tree_t), intent(in) :: parse_tree
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      logical :: short
      u = output_unit (unit);  if (u < 0)  return
      short = .true.;  if (present (verbose))  short = .not. verbose
      write (u, "(A)") "Parse tree:"
      if (associated (parse_tree%root_node)) then
        call parse_node_write_rec (parse_tree%root_node, unit, short, 1)
      else
        write (u, *) "[empty]"
      end if
    end subroutine parse_tree_write
```

This is a generic error that can be issued if the parse tree does not meet the
expectaions of the parser. This most certainly indicates a bug.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_bug
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_bug (node, keys)
  type(parse_node_t), intent(in) :: node
  character(*), intent(in) :: keys
  call parse_node_write (node)
  call msg_bug (" Inconsistency in parse tree: expected " // keys)
end subroutine parse_tree_bug
```

3.4.7   Access the parser

For scanning the parse tree we give access to the top node, as a node pointer.
Of course, there should be no write access.

⟨*Parser: public*⟩+≡

```
public :: parse_tree_get_root_ptr
```

⟨*Parser: procedures*⟩+≡

```
function parse_tree_get_root_ptr (parse_tree) result (node)
  type(parse_node_t), pointer :: node
  type(parse_tree_t), intent(in), target :: parse_tree
  node => parse_tree%root_node
end function parse_tree_get_root_ptr
```

3.4.8   Tools

This operation traverses the parse tree and simplifies any occurences of a set of
syntax rules. If such a parse node has only one sub-node, it is replaced by that
subnode. (This makes sense only of the rules to eliminate have no meaningful
token.)

⟨*Parser: public*⟩+≡

```
public :: parse_tree_reduce
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_reduce (parse_tree, rule_key)
  type(parse_tree_t), intent(inout) :: parse_tree
  type(string_t), dimension(:), intent(in) :: rule_key
  type(parse_node_t), pointer :: pn
  pn => parse_tree%root_node
  if (associated (pn)) then
     call parse_node_reduce (pn, null(), null())
  end if
contains
  recursive subroutine parse_node_reduce (pn, pn_prev, pn_parent)
    type(parse_node_t), intent(inout), pointer :: pn
    type(parse_node_t), intent(in), pointer :: pn_prev, pn_parent
    type(parse_node_t), pointer :: pn_sub, pn_sub_prev, pn_tmp
    pn_sub_prev => null ()
    pn_sub => pn%sub_first
    do while (associated (pn_sub))
       call parse_node_reduce (pn_sub, pn_sub_prev, pn)
       pn_sub_prev => pn_sub
       pn_sub => pn_sub%next
```

```
            end do
         if (parse_node_get_n_sub (pn) == 1) then
            if (matches (parse_node_get_rule_key (pn), rule_key)) then
               pn_tmp => pn
               pn => pn%sub_first
               if (associated (pn_prev)) then
                  pn_prev%next => pn
               else if (associated (pn_parent)) then
                  pn_parent%sub_first => pn
               else
                  parse_tree%root_node => pn
               end if
               if (associated (pn_tmp%next)) then
                  pn%next => pn_tmp%next
               else if (associated (pn_parent)) then
                  pn_parent%sub_last => pn
               end if
               call parse_node_final (pn_tmp, recursive=.false.)
               deallocate (pn_tmp)
            end if
         end if
      end subroutine parse_node_reduce
      function matches (key, key_list) result (flag)
        logical :: flag
        type(string_t), intent(in) :: key
        type(string_t), dimension(:), intent(in) :: key_list
        integer :: i
        flag = .true.
        do i = 1, size (key_list)
           if (key == key_list(i))  return
        end do
        flag = .false.
      end function matches
   end subroutine parse_tree_reduce
```

3.4.9   Applications

For a file of the form

```
process foo, bar
  <something>
process xyz
  <something>
```

get the `<something>` entry node for the first matching process tag. If no matching entry is found, the node pointer remains unassociated.

⟨*Parser: public*⟩+≡
```
    public :: parse_tree_get_process_ptr
```

⟨*Parser: procedures*⟩+≡
```
    function parse_tree_get_process_ptr (parse_tree, process) result (node)
      type(parse_node_t), pointer :: node
      type(parse_tree_t), intent(in), target :: parse_tree
```

```
      type(string_t), intent(in) :: process
      type(parse_node_t), pointer :: node_root, node_process_def
      type(parse_node_t), pointer :: node_process_phs, node_process_list
      integer :: j
      node_root => parse_tree_get_root_ptr (parse_tree)
      if (associated (node_root)) then
         node_process_phs => parse_node_get_sub_ptr (node_root)
         SCAN_FILE: do while (associated (node_process_phs))
            node_process_def => parse_node_get_sub_ptr (node_process_phs)
            node_process_list => parse_node_get_sub_ptr (node_process_def, 2)
            do j = 1, parse_node_get_n_sub (node_process_list)
               if (parse_node_get_string &
                     (parse_node_get_sub_ptr (node_process_list, j)) &
                     == process) then
                  node => parse_node_get_next_ptr (node_process_def)
                  return
               end if
            end do
            node_process_phs => parse_node_get_next_ptr (node_process_phs)
         end do SCAN_FILE
         node => null ()
      else
         node => null ()
      end if
    end function parse_tree_get_process_ptr
```

3.4.10 Unit tests

⟨*Parser: public*⟩+≡
```
    public :: parse_test
```

⟨*Parser: procedures*⟩+≡
```
    subroutine parse_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
    ⟨*Parser: execute tests*⟩
```
    end subroutine parse_test
```

This checks the parser.

⟨*Parser: execute tests*⟩≡
```
    call test (parse_1, "parse_1", &
         "check the parser", &
         u, results)
```

⟨*Parser: tests*⟩≡
```
    subroutine parse_1 (u)
      use ifiles
      use lexers
      integer, intent(in) :: u

      type(ifile_t) :: ifile
      type(syntax_t), target :: syntax
      type(lexer_t) :: lexer
```

```
type(stream_t), target :: stream
type(parse_tree_t), target :: parse_tree

write (u, "(A)")  "* Test output: Parsing"
write (u, "(A)")  "*   Purpose: test parse routines"
write (u, "(A)")

call ifile_append (ifile, "SEQ expr = term addition*")
call ifile_append (ifile, "SEQ addition = plus_or_minus term")
call ifile_append (ifile, "SEQ term = factor multiplication*")
call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
call ifile_append (ifile, "SEQ factor = atom exponentiation*")
call ifile_append (ifile, "SEQ exponentiation = '^' atom")
call ifile_append (ifile, "ALT atom = real | delimited_expr")
call ifile_append (ifile, "GRO delimited_expr = ( expr )")
call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
call ifile_append (ifile, "KEY '+'")
call ifile_append (ifile, "KEY '-'")
call ifile_append (ifile, "KEY '*'")
call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "KEY '^'")
call ifile_append (ifile, "REA real")

write (u, "(A)")  "* File contents (syntax definition):"
call ifile_write (ifile, u)
write (u, "(A)")  "EOF"
write (u, "(A)")

call syntax_init (syntax, ifile)
call ifile_final (ifile)
call syntax_write (syntax, u)
write (u, "(A)")

call lexer_init (lexer, &
     comment_chars = "", &
     quote_chars = "'", &
     quote_match = "'", &
     single_chars = "+-*/^()", &
     special_class = [""] , &
     keyword_list = syntax_get_keyword_list_ptr (syntax))
call lexer_write_setup (lexer, u)
write (u, "(A)")

call ifile_append (ifile, "(27+8^3-2/3)*(4+7)^2*99")
write (u, "(A)")  "* File contents (input file):"
call ifile_write (ifile, u)
write (u, "(A)")  "EOF"
print *

call stream_init (stream, ifile)
call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax, lexer)
call stream_final (stream)
```

211

```
      call parse_tree_write (parse_tree, u, .true.)
      print *

      write (u, "(A)")  "* Cleanup, everything should now be empty:"
      write (u, "(A)")

      call parse_tree_final (parse_tree)
      call parse_tree_write (parse_tree, u, .true.)
      write (u, "(A)")

      call lexer_final (lexer)
      call lexer_write_setup (lexer, u)
      write (u, "(A)")

      call ifile_final (ifile)
      write (u, "(A)")  "* File contents:"
      call ifile_write (ifile, u)
      write (u, "(A)")  "EOF"
      write (u, "(A)")

      call syntax_final (syntax)
      call syntax_write (syntax, u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: parser_1"

    end subroutine parse_1
```

3.5 XML Parser

The XML parser is actually independent from the previous modules of lexer and parser. However, for a generic I/O interface we make use of the `stream_t` facility.

We need the XML parser for reading and writing LHEF data files. Only a subset of XML is actually relevant. The parser is of the "pull" type, i.e., the program steers the reading of XML data in a context-sensitive manner.

⟨xml.f90⟩≡
  ⟨*File header*⟩

  module xml

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: BLANK, TAB !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use ifiles
    use lexers

  ⟨*Standard module head*⟩

  ⟨*XML: public*⟩

  ⟨*XML: types*⟩

  contains

  ⟨*XML: procedures*⟩

  ⟨*XML: tests*⟩

  end module xml

3.5.1 Cached Stream

The stream type as defined in the `lexer` module is versatile regarding the choice of input channel, but it does not allow reading a section more than once. Here, we define an extension where we can return a string to the stream, which is stored in a cache variable, and presented to the caller for the next read.

⟨*XML: public*⟩≡
    public :: cstream_t

⟨*XML: types*⟩≡
    type, extends (stream_t) :: cstream_t
       logical :: cache_is_empty = .true.
       type(string_t) :: cache
     contains
     ⟨*XML: cstream: TBP*⟩
    end type cstream_t

213

The initializers are simply inherited. We require a generic.

⟨*XML: cstream: TBP*⟩≡
```
generic :: init => init_filename, init_unit, init_string, &
    init_ifile, init_line
procedure :: init_filename => cstream_init_filename
procedure :: init_unit => cstream_init_unit
procedure :: init_string => cstream_init_string
procedure :: init_ifile => cstream_init_ifile
procedure :: init_line => cstream_init_line
```

⟨*XML: procedures*⟩≡
```
subroutine cstream_init_filename (cstream, filename)
  class(cstream_t), intent(out) :: cstream
  character(*), intent(in) :: filename
  call stream_init (cstream%stream_t, filename)
end subroutine cstream_init_filename

subroutine cstream_init_unit (cstream, unit)
  class(cstream_t), intent(out) :: cstream
  integer, intent(in) :: unit
  call stream_init (cstream%stream_t, unit)
end subroutine cstream_init_unit

subroutine cstream_init_string (cstream, string)
  class(cstream_t), intent(out) :: cstream
  type(string_t), intent(in) :: string
  call stream_init (cstream%stream_t, string)
end subroutine cstream_init_string

subroutine cstream_init_ifile (cstream, ifile)
  class(cstream_t), intent(out) :: cstream
  type(ifile_t), intent(in) :: ifile
  call stream_init (cstream%stream_t, ifile)
end subroutine cstream_init_ifile

subroutine cstream_init_line (cstream, line)
  class(cstream_t), intent(out) :: cstream
  type(line_p), intent(in) :: line
  call stream_init (cstream%stream_t, line)
end subroutine cstream_init_line
```

Finalizer: also inherited

⟨*XML: cstream: TBP*⟩+≡
```
procedure :: final => cstream_final
```

⟨*XML: procedures*⟩+≡
```
subroutine cstream_final (cstream)
  class(cstream_t), intent(inout) :: cstream
  cstream%cache_is_empty = .true.
  call stream_final (cstream%stream_t)
end subroutine cstream_final
```

Get record: now, if there is a cache string, return this instead of the record from the stream.

214

⟨XML: cstream: TBP⟩+≡

```
    procedure :: get_record => cstream_get_record
```

⟨XML: procedures⟩+≡

```
    subroutine cstream_get_record (cstream, string, iostat)
      class(cstream_t), intent(inout) :: cstream
      type(string_t), intent(out) :: string
      integer, intent(out) :: iostat
      if (cstream%cache_is_empty) then
         call stream_get_record (cstream%stream_t, string, iostat)
      else
         string = cstream%cache
         cstream%cache_is_empty = .true.
         iostat = 0
      end if
    end subroutine cstream_get_record
```

Revert: return the (partially read) record to the stream, putting it in the cache.

⟨XML: cstream: TBP⟩+≡

```
    procedure :: revert_record => cstream_revert_record
```

⟨XML: procedures⟩+≡

```
    subroutine cstream_revert_record (cstream, string)
      class(cstream_t), intent(inout) :: cstream
      type(string_t), intent(in) :: string
      if (cstream%cache_is_empty) then
         cstream%cache = string
         cstream%cache_is_empty = .false.
      else
         call msg_bug ("CStream: attempt to revert twice")
      end if
    end subroutine cstream_revert_record
```

3.5.2 Attributes

A tag attribute has a name and a value; both are strings. When the attribute is defined, the known flag indicates this.

⟨XML: types⟩+≡

```
    type :: attribute_t
      type(string_t) :: name
      type(string_t) :: value
      logical :: known = .false.
     contains
     ⟨XML: attribute: TBP⟩
    end type attribute_t
```

Output in standard format, non-advancing. (If the value is unknown, we indicate this by a question mark, which is non-standard.)

⟨XML: attribute: TBP⟩≡

```
    procedure :: write => attribute_write
```

⟨XML: procedures⟩+≡
```
    subroutine attribute_write (object, unit)
      class(attribute_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(A,'=')", advance = "no")  char (object%name)
      if (object%known) then
          write (u, "(A,A,A)", advance = "no")  '"', char (object%value), '"'
      else
          write (u, "('?')", advance = "no")
      end if
    end subroutine attribute_write
```

This is a genuine constructor. The value is optional.

⟨XML: public⟩+≡
```
    public :: xml_attribute
```

⟨XML: procedures⟩+≡
```
    function xml_attribute (name, value) result (attribute)
      type(string_t), intent(in) :: name
      type(string_t), intent(in), optional :: value
      type(attribute_t) :: attribute
      attribute%name = name
      if (present (value)) then
          attribute%value = value
          attribute%known = .true.
      else
          attribute%known = .false.
      end if
    end function xml_attribute
```

Set a value explicitly.

⟨XML: attribute: TBP⟩+≡
```
    procedure :: set_value => attribute_set_value
```

⟨XML: procedures⟩+≡
```
    subroutine attribute_set_value (attribute, value)
      class(attribute_t), intent(inout) :: attribute
      type(string_t), intent(in) :: value
      attribute%value = value
      attribute%known = .true.
    end subroutine attribute_set_value
```

Extract a value. If unset, return "?"

⟨XML: attribute: TBP⟩+≡
```
    procedure :: get_value => attribute_get_value
```

⟨XML: procedures⟩+≡
```
    function attribute_get_value (attribute) result (value)
      class(attribute_t), intent(in) :: attribute
      type(string_t) :: value
      if (attribute%known) then
          value = attribute%value
```

```
        else
            value = "?"
        end if
    end function attribute_get_value
```

3.5.3 The Tag Type

The basic entity is the internal representation of an XML tag. The tag has a name, a well-defined set of attributes which may be mandatory or optional, and a flag that determines whether there is content or not. The content itself is not stored in the data structure.

⟨*XML: public*⟩+≡
```
    public :: xml_tag_t
```

⟨*XML: types*⟩+≡
```
    type :: xml_tag_t
      type(string_t) :: name
      type(attribute_t), dimension(:), allocatable :: attribute
      logical :: has_content = .false.
    contains
    ⟨XML: tag: TBP⟩
    end type xml_tag_t
```

Initialization. There are different versions, depending on content.

⟨*XML: tag: TBP*⟩≡
```
    generic :: init => init_no_attributes
    procedure :: init_no_attributes => tag_init_no_attributes
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_init_no_attributes (tag, name, has_content)
      class(xml_tag_t), intent(out) :: tag
      type(string_t), intent(in) :: name
      logical, intent(in), optional :: has_content
      tag%name = name
      allocate (tag%attribute (0))
      if (present (has_content))  tag%has_content = has_content
    end subroutine tag_init_no_attributes
```

This version sets attributes.

⟨*XML: tag: TBP*⟩+≡
```
    generic :: init => init_with_attributes
    procedure :: init_with_attributes => tag_init_with_attributes
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_init_with_attributes (tag, name, attribute, has_content)
      class(xml_tag_t), intent(out) :: tag
      type(string_t), intent(in) :: name
      type(attribute_t), dimension(:), intent(in) :: attribute
      logical, intent(in), optional :: has_content
      tag%name = name
      allocate (tag%attribute (size (attribute)))
      tag%attribute = attribute
```

```
      if (present (has_content))  tag%has_content = has_content
    end subroutine tag_init_with_attributes
```

Set an attribute value explicitly.
⟨*XML: tag: TBP*⟩+≡
```
    procedure :: set_attribute => tag_set_attribute
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_set_attribute (tag, i, value)
      class(xml_tag_t), intent(inout) :: tag
      integer, intent(in) :: i
      type(string_t), intent(in) :: value
      call tag%attribute(i)%set_value (value)
    end subroutine tag_set_attribute
```

Get an attribute value.
⟨*XML: tag: TBP*⟩+≡
```
    procedure :: get_attribute => tag_get_attribute
```

⟨*XML: procedures*⟩+≡
```
    function tag_get_attribute (tag, i) result (value)
      class(xml_tag_t), intent(in) :: tag
      integer, intent(in) :: i
      type(string_t) :: value
      value = tag%attribute(i)%get_value ()
    end function tag_get_attribute
```

Output to an I/O unit, default STDOUT. We use non-advancing output.
⟨*XML: tag: TBP*⟩+≡
```
    generic :: write => write_without_content
    procedure :: write_without_content => tag_write
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_write (tag, unit)
      class(xml_tag_t), intent(in) :: tag
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "('<',A)", advance = "no")  char (tag%name)
      do i = 1, size (tag%attribute)
         write (u, "(1x)", advance = "no")
         call tag%attribute(i)%write (u)
      end do
      if (tag%has_content) then
         write (u, "('>')", advance = "no")
      else
         write (u, "(' />')", advance = "no")
      end if
    end subroutine tag_write
```

If there is content, we should write the context next (arbitrary format), the write the corresponding closing tag. Again, non-advancing.
⟨*XML: tag: TBP*⟩+≡
```
    procedure :: close => tag_close
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_close (tag, unit)
      class(xml_tag_t), intent(in) :: tag
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "('</',A,'>')", advance = "no")  char (tag%name)
    end subroutine tag_close
```

Given content as a single string, we can write tag, content, and closing at once

⟨*XML: tag: TBP*⟩+≡
```
    generic :: write => write_with_content
    procedure :: write_with_content => tag_write_with_content
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_write_with_content (tag, content, unit)
      class(xml_tag_t), intent(in) :: tag
      type(string_t), intent(in) :: content
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      call tag%write (u)
      write (u, "(A)", advance = "no")  char (content)
      call tag%close (u)
    end subroutine tag_write_with_content
```

Input from stream. We know what we are looking for, so we check if the name matches, then fill attributes. We report an error if (a) an I/O error occurs, (b) we reach EOF before encountering the tag, (c) if the tag is incomplete.

Trailing text after reading a tag is put back to the input stream.

We assume that the tag is not broken across records, and that there is only one tag within the record. This is more restricted than standard XML.

⟨*XML: tag: TBP*⟩+≡
```
    procedure :: read => tag_read
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_read (tag, cstream, success)
      class(xml_tag_t), intent(inout) :: tag
      type(cstream_t), intent(inout) :: cstream
      logical, intent(out) :: success
      type(string_t) :: string
      integer :: iostat, p1, p2
      character(2), parameter :: WS = BLANK // TAB
      logical :: done

      ! Skip comments and blank lines
      FIND_NON_COMMENT: do
         FIND_NONEMPTY_RECORD: do
            call cstream%get_record (string, iostat)
            if (iostat /= 0)  call err_io ()
            p1 = verify (string, WS)
            if (p1 > 0)  exit FIND_NONEMPTY_RECORD
         end do FIND_NONEMPTY_RECORD
```

```
      ! Look for comment beginning
      p2 = p1 + 3
      if (extract (string, p1, p2) /= "<!--")  exit FIND_NON_COMMENT

      ! Look for comment end, then restart
      string = extract (string, p2 + 1)
      FIND_COMMENT_END: do
         do p1 = 1, len (string) - 2
            p2 = p1 + 2
            if (extract (string, p1, p2) == "-->") then

               ! Return trailing text to the stream
               string = extract (string, p2 + 1)
               if (string /= "")  call cstream%revert_record (string)
               exit FIND_COMMENT_END

            end if
         end do
         call cstream%get_record (string, iostat)
         if (iostat /= 0)  call err_io ()
      end do FIND_COMMENT_END
end do FIND_NON_COMMENT

! Look for opening <
p2 = p1
if (extract (string, p1, p2) /= "<") then
   call cstream%revert_record (string)
   success = .false.;  return
else

   ! Look for tag name
   string = extract (string, p2 + 1)
   p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
   p2 = p1 + len (tag%name) - 1
   if (extract (string, p1, p2) /= tag%name) then
      call cstream%revert_record ("<" // string)
      success = .false.;  return
   else

      ! Look for attributes
      string = extract (string, p2 + 1)
      READ_ATTRIBUTES: do
         call tag%read_attribute (string, done)
         if (done)  exit READ_ATTRIBUTES
      end do READ_ATTRIBUTES

      ! Look for closing >
      p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
      p2 = p1
      if (extract (string, p1, p1) == ">") then
         tag%has_content = .true.
      else
```

```
           ! Look for closing />
           p2 = p1 + 1
           if (extract (string, p1, p2) /= "/>")  call err_incomplete ()
        end if

        ! Return trailing text to the stream
        string = extract (string, p2 + 1)
        if (string /= "")  call cstream%revert_record (string)
        success = .true.

      end if
    end if

  contains

    subroutine err_io ()
      select case (iostat)
      case (:-1)
         call msg_fatal ("XML: Error reading tag '" // char (tag%name) &
              // "': end of file")
      case (1:)
         call msg_fatal ("XML: Error reading tag '" // char (tag%name) &
              // "': I/O error")
      end select
      success = .false.
    end subroutine err_io

    subroutine err_incomplete ()
      call msg_fatal ("XML: Error reading tag '" // char (tag%name) &
           // "': tag incomplete")
      success = .false.
    end subroutine err_incomplete

  end subroutine tag_read
```

Read a single attribute. If the attribute is valid, assign the value. Setting a value twice should be an error, but is not detected. If the attribute is unknown, ignore it. If we reach the closing sign, report this.

⟨*XML: tag: TBP*⟩+≡
```
    procedure :: read_attribute => tag_read_attribute
```

⟨*XML: procedures*⟩+≡
```
    subroutine tag_read_attribute (tag, string, done)
      class(xml_tag_t), intent(inout) :: tag
      type(string_t), intent(inout) :: string
      logical, intent(out) :: done
      character(2), parameter :: WS = BLANK // TAB
      type(string_t) :: name, value
      integer :: p1, p2, i

      p1 = verify (string, WS);  if (p1 == 0)  call err ()
      p2 = p1

      ! Look for first terminating '>' or '/>'
```

221

```
         if (extract (string, p1, p2) == ">") then
            done = .true.
         else
            p2 = p1 + 1
            if (extract (string, p1, p2) == "/>") then
               done = .true.
            else

               ! Look for '='
               p2 = scan (string, '=')
               if (p2 == 0)  call err ()
               name = trim (extract (string, p1, p2 - 1))

               ! Look for '"'
               string = extract (string, p2 + 1)
               p1 = verify (string, WS);  if (p1 == 0)  call err ()
               p2 = p1
               if (extract (string, p1, p2) /= '"')  call err ()

               ! Look for matching '"' and get value
               string = extract (string, p2 + 1)
               p1 = 1
               p2 = scan (string, '"')
               if (p2 == 0)  call err ()
               value = extract (string, p1, p2 - 1)

               SCAN_KNOWN_ATTRIBUTES: do i = 1, size (tag%attribute)
                  if (name == tag%attribute(i)%name) then
                     call tag%attribute(i)%set_value (value)
                     exit SCAN_KNOWN_ATTRIBUTES
                  end if
               end do SCAN_KNOWN_ATTRIBUTES

               string = extract (string, p2 + 1)
               done = .false.
            end if
         end if

    contains

      subroutine err ()
        call msg_fatal ("XML: Error reading attributes of '" // char (tag%name) &
             // "': syntax error")
      end subroutine err

    end subroutine tag_read_attribute
```

Read the content string of a tag. We check for the appropriate closing tag and report it. If a closing tag does not match in name, ignore it.

Note: this assumes that no tag with the same name is embedded in the current content. Also, we do not check for XML validity inside the content.

⟨*XML: tag: TBP*⟩+≡

```
    procedure :: read_content => tag_read_content
```

⟨XML: procedures⟩+≡

```fortran
    subroutine tag_read_content (tag, cstream, content, closing)
      class(xml_tag_t), intent(in) :: tag
      type(cstream_t), intent(inout) :: cstream
      type(string_t), intent(out) :: content
      type(string_t) :: string
      logical, intent(out) :: closing
      integer :: iostat
      integer :: p0, p1, p2
      character(2), parameter :: WS = BLANK // TAB
      call cstream%get_record (content, iostat)
      if (iostat /= 0)  call err_io ()
      closing = .false.
      FIND_CLOSING: do p0 = 1, len (content) - 1

         ! Look for terminating </
         p1 = p0
         p2 = p1 + 1
         if (extract (content, p1, p2) == "</") then

            ! Look for closing tag name
            string = extract (content, p2 + 1)
            p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
            p2 = p1 + len (tag%name) - 1
            if (extract (string, p1, p2) == tag%name) then

               ! Tag name matches: look for final >
               string = extract (string, p2 + 1)
               p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
               p2 = p1
               if (extract (string, p1, p2) /= ">")  call err_incomplete ()

               ! Return trailing text to the stream
               string = extract (string, p2 + 1)
               if (string /= "")  call cstream%revert_record (string)
               content = extract (content, 1, p0 -1)
               closing = .true.
               exit FIND_CLOSING

            end if
         end if
      end do FIND_CLOSING

   contains

      subroutine err_io ()
        select case (iostat)
        case (:-1)
           call msg_fatal ("XML: Error reading content of '" // char (tag%name) &
                  // "': end of file")
        case (1:)
           call msg_fatal ("XML: Error reading content of '" // char (tag%name) &
                  // "': I/O error")
        end select
```

```
      closing = .false.
    end subroutine err_io

    subroutine err_incomplete ()
      call msg_fatal ("XML: Error reading content '" // char (tag%name) &
            // "': closing tag incomplete")
      closing = .false.
    end subroutine err_incomplete

  end subroutine tag_read_content
```

3.5.4   Unit tests

⟨*XML: public*⟩+≡
```
    public :: xml_test
```

⟨*XML: tests*⟩≡
```
    subroutine xml_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
      ⟨*XML: execute tests*⟩
```
  end subroutine xml_test
```

3.5.5   Auxiliary Routines

Show the contents of a temporary file, i.e., open unit.

⟨*XML: tests*⟩+≡
```
    subroutine show (u_tmp, u)
      integer, intent(in) :: u_tmp, u
      character (80) :: buffer
      integer :: iostat
      write (u, "(A)")  "File content:"
      rewind (u_tmp)
      do
         read (u_tmp, "(A)", iostat = iostat)  buffer
         if (iostat /= 0)  exit
         write (u, "(A)")  trim (buffer)
      end do
      rewind (u_tmp)
    end subroutine show
```

3.5.6   Basic Tag I/O

Write a tag and read it again, using a temporary file.

⟨*XML: execute tests*⟩≡
```
    call test (xml_1, "xml_1", &
        "basic I/O", &
        u, results)
```

⟨*XML: tests*⟩+≡

```fortran
    subroutine xml_1 (u)
      integer, intent(in) :: u
      type(xml_tag_t), allocatable :: tag
      integer :: u_tmp
      type(cstream_t) :: cstream
      logical :: success

      write (u, "(A)")  "* Test output: xml_1"
      write (u, "(A)")  "*   Purpose: write and read tag"
      write (u, "(A)")

      write (u, "(A)")  "* Empty tag"
      write (u, *)

      u_tmp = free_unit ()
      open (u_tmp, status = "scratch", action = "readwrite")

      allocate (tag)
      call tag%init (var_str ("tagname"))
      call tag%write (u_tmp)
      write (u_tmp, *)
      deallocate (tag)

      call show (u_tmp, u)

      write (u, *)
      write (u, "(A)")  "Result from read:"
      call cstream%init (u_tmp)
      allocate (tag)
      call tag%init (var_str ("tagname"))
      call tag%read (cstream, success)
      call tag%write (u)
      write (u, *)
      write (u, "(A,L1)")  "success = ", success
      deallocate (tag)
      call cstream%final ()

      write (u, *)
      write (u, "(A)")  "* Tag with preceding blank lines"
      write (u, *)

      u_tmp = free_unit ()
      open (u_tmp, status = "scratch", action = "readwrite")

      allocate (tag)
      call tag%init (var_str ("tagname"))
      write (u_tmp, *)
      write (u_tmp, "(A)")  "    "
      write (u_tmp, "(2x)", advance = "no")
      call tag%write (u_tmp)
      write (u_tmp, *)
      deallocate (tag)
```

```
call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
deallocate (tag)
call cstream%final ()

write (u, *)
write (u, "(A)")  "* Tag with preceding comments"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag)
call tag%init (var_str ("tagname"))
write (u_tmp, "(A)")  "<!-- comment -->"
write (u_tmp, *)
write (u_tmp, "(A)")  "<!-- multiline"
write (u_tmp, "(A)")  "    comment -->"
call tag%write (u_tmp)
write (u_tmp, *)
deallocate (tag)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
deallocate (tag)
call cstream%final ()

write (u, *)
write (u, "(A)")  "* Tag with name mismatch"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag)
call tag%init (var_str ("wrongname"))
```

```
    call tag%write (u_tmp)
    write (u_tmp, *)
    deallocate (tag)

    call show (u_tmp, u)

    write (u, *)
    write (u, "(A)")  "Result from read:"
    call cstream%init (u_tmp)
    allocate (tag)
    call tag%init (var_str ("tagname"))
    call tag%read (cstream, success)
    call tag%write (u)
    write (u, *)
    write (u, "(A,L1)")  "success = ", success
    deallocate (tag)
    call cstream%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: xml_1"

  end subroutine xml_1
```

3.5.7 Optional Tag

Write and read two tags, one of them optional.

⟨*XML: execute tests*⟩+≡
```
    call test (xml_2, "xml_2", &
         "optional tag", &
         u, results)
```

⟨*XML: tests*⟩+≡
```
    subroutine xml_2 (u)
      integer, intent(in) :: u
      type(xml_tag_t), allocatable :: tag1, tag2
      integer :: u_tmp
      type(cstream_t) :: cstream
      logical :: success

      write (u, "(A)")  "* Test output: xml_2"
      write (u, "(A)")  "*   Purpose: handle optional tag"
      write (u, "(A)")

      write (u, "(A)")  "* Optional tag present"
      write (u, *)

      u_tmp = free_unit ()
      open (u_tmp, status = "scratch", action = "readwrite")

      allocate (tag1)
      call tag1%init (var_str ("option"))
      call tag1%write (u_tmp)
      write (u_tmp, *)
```

```
allocate (tag2)
call tag2%init (var_str ("tagname"))
call tag2%write (u_tmp)
write (u_tmp, *)
deallocate (tag1, tag2)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag1)
call tag1%init (var_str ("option"))
call tag1%read (cstream, success)
call tag1%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, *)
allocate (tag2)
call tag2%init (var_str ("tagname"))
call tag2%read (cstream, success)
call tag2%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
deallocate (tag1, tag2)
call cstream%final ()

write (u, *)
write (u, "(A)")  "* Optional tag absent"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag2)
call tag2%init (var_str ("tagname"))
call tag2%write (u_tmp)
write (u_tmp, *)
deallocate (tag2)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag1)
call tag1%init (var_str ("option"))
call tag1%read (cstream, success)
call tag1%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, *)
allocate (tag2)
call tag2%init (var_str ("tagname"))
```

```
    call tag2%read (cstream, success)
    call tag2%write (u)
    write (u, *)
    write (u, "(A,L1)")  "success = ", success
    deallocate (tag1, tag2)
    call cstream%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: xml_2"

end subroutine xml_2
```

3.5.8 Optional Tag

Write and read a tag with single-line content.

⟨*XML: execute tests*⟩+≡
```
    call test (xml_3, "xml_3", &
        "content", &
        u, results)
```
⟨*XML: tests*⟩+≡
```
subroutine xml_3 (u)
  integer, intent(in) :: u
  type(xml_tag_t), allocatable :: tag
  integer :: u_tmp
  type(cstream_t) :: cstream
  logical :: success, closing
  type(string_t) :: content

  write (u, "(A)")  "* Test output: xml_3"
  write (u, "(A)")  "*   Purpose: handle tag with content"
  write (u, "(A)")

  write (u, "(A)")  "* Tag without content"
  write (u, *)

  u_tmp = free_unit ()
  open (u_tmp, status = "scratch", action = "readwrite")

  allocate (tag)
  call tag%init (var_str ("tagname"))
  call tag%write (u_tmp)
  write (u_tmp, *)
  deallocate (tag)

  call show (u_tmp, u)

  write (u, *)
  write (u, "(A)")  "Result from read:"
  call cstream%init (u_tmp)
  allocate (tag)
  call tag%init (var_str ("tagname"))
  call tag%read (cstream, success)
```

```
call tag%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, "(A,L1)")  "content = ", tag%has_content
write (u, *)
deallocate (tag)
call cstream%final ()

write (u, "(A)")  "* Tag with content"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag)
call tag%init (var_str ("tagname"), has_content = .true.)
call tag%write (var_str ("Content text"), u_tmp)
write (u_tmp, *)
deallocate (tag)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%read_content (cstream, content, closing)
call tag%write (u)
write (u, "(A)", advance = "no")  char (content)
call tag%close (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, "(A,L1)")  "content = ", tag%has_content
write (u, "(A,L1)")  "closing = ", closing
deallocate (tag)
call cstream%final ()

write (u, *)
write (u, "(A)")  "* Tag with multiline content"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag)
call tag%init (var_str ("tagname"), has_content = .true.)
call tag%write (u_tmp)
write (u_tmp, *)
write (u_tmp, "(A)")  "Line 1"
write (u_tmp, "(A)")  "Line 2"
call tag%close (u_tmp)
write (u_tmp, *)
```

```
      deallocate (tag)

      call show (u_tmp, u)

      write (u, *)
      write (u, "(A)")  "Result from read:"
      call cstream%init (u_tmp)
      allocate (tag)
      call tag%init (var_str ("tagname"))
      call tag%read (cstream, success)
      call tag%write (u)
      write (u, *)
      do
         call tag%read_content (cstream, content, closing)
         if (closing)  exit
         write (u, "(A)")  char (content)
      end do
      call tag%close (u)
      write (u, *)
      write (u, "(A,L1)")  "success = ", success
      write (u, "(A,L1)")  "content = ", tag%has_content
      deallocate (tag)
      call cstream%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: xml_3"

    end subroutine xml_3
```

3.5.9 Basic Tag I/O

Write a tag and read it again, using a temporary file.

⟨*XML: execute tests*⟩+≡
```
    call test (xml_4, "xml_4", &
         "attributes", &
         u, results)
```

⟨*XML: tests*⟩+≡
```
    subroutine xml_4 (u)
      integer, intent(in) :: u
      type(xml_tag_t), allocatable :: tag
      integer :: u_tmp
      type(cstream_t) :: cstream
      logical :: success

      write (u, "(A)")  "* Test output: xml_4"
      write (u, "(A)")  "*   Purpose: handle tag with attributes"
      write (u, "(A)")

      write (u, "(A)")  "* Tag with one mandatory and one optional attribute,"
      write (u, "(A)")  "* unknown attribute ignored"
      write (u, *)
```

```
    u_tmp = free_unit ()
    open (u_tmp, status = "scratch", action = "readwrite")

    allocate (tag)
    call tag%init (var_str ("tagname"), &
         [xml_attribute (var_str ("a1"), var_str ("foo")), &
          xml_attribute (var_str ("a3"), var_str ("gee"))])
    call tag%write (u_tmp)
    deallocate (tag)

    call show (u_tmp, u)

    write (u, *)
    write (u, "(A)")  "Result from read:"
    call cstream%init (u_tmp)
    allocate (tag)
    call tag%init (var_str ("tagname"), &
         [xml_attribute (var_str ("a1")), &
          xml_attribute (var_str ("a2"), var_str ("bar"))])
    call tag%read (cstream, success)
    call tag%write (u)
    write (u, *)
    deallocate (tag)
    call cstream%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: xml_4"

  end subroutine xml_4
```

Chapter 4

Physics library

This part consists of two modules:

constants Physical and mathematical parameters that never change. The file has been moved to the `src/misc` subdirectory of the `WHIZARD` project.

c_particles A simple data type for particles which is C compatible.

lorentz Define three-vectors, four-vectors and Lorentz transformations and common operations for them.

sm_physics Here, running functions are stored for special kinematical setup like running coupling constants, Catani-Seymour dipoles, or Sudakov factors.

4.1   C-compatible Particle Type

For easy communication with C code, we introduce a simple C-compatible type for particles. It has the contents of the `prt_t` type defined in the `subevents` module below (except the source information).

The `c_prt` type is transparent, and its contents should be regarded as part of the interface.

⟨c_particles.f90⟩≡
  ⟨*File header*⟩

```
  module c_particles

    use iso_c_binding !NODEP!
```
⟨*Use file utils*⟩
```
    use limits, only: FMT_14, FMT_19 !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*C Particles: public*⟩

  ⟨*C Particles: types*⟩

```
  contains
```

  ⟨*C Particles: procedures*⟩
```
  end module c_particles
```

⟨*C Particles: public*⟩≡
```
    public :: c_prt_t
```

⟨*C Particles: types*⟩≡
```
    type, bind(C) :: c_prt_t
       integer(c_int) :: type = 0
       integer(c_int) :: pdg = 0
       integer(c_int) :: polarized = 0
       integer(c_int) :: h = 0
       real(c_double) :: pe = 0
       real(c_double) :: px = 0
       real(c_double) :: py = 0
       real(c_double) :: pz = 0
       real(c_double) :: p2 = 0
    end type c_prt_t
```

This is for debugging only, there is no C binding. It is a simplified version of `prt_write`.

⟨*C Particles: public*⟩+≡
```
    public :: c_prt_write
```

⟨*C Particles: procedures*⟩≡
```
    subroutine c_prt_write (prt, unit)
      type(c_prt_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
```

```
   write (u, "(1x,A)", advance="no")  "prt("
   write (u, "(I0,':')", advance="no")  prt%type
   if (prt%polarized /= 0) then
      write (u, "(I0,'/',I0,'|')", advance="no")  prt%pdg, prt%h
   else
      write (u, "(I0,'|')", advance="no") prt%pdg
   end if
   write (u, "(" // FMT_14 // ",';'," // FMT_14 // ",',',' // &
        FMT_14 // ",',',' // FMT_14 // ")", advance="no") &
        prt%pe, prt%px, prt%py, prt%pz
   write (u, "('|'," // FMT_19 // ")", advance="no")  prt%p2
   write (u, "(A)")  ")"
end subroutine c_prt_write
```

4.2 Lorentz algebra

Define Lorentz vectors, three-vectors, boosts, and some functions to manipulate them.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

⟨`lorentz.f90`⟩≡
  ⟨*File header*⟩

```
module lorentz
```

  ⟨*Use kinds*⟩
```
    use constants, only: pi, twopi, degree !NODEP!
```
  ⟨*Use file utils*⟩
```
    use limits, only: FMT_15, FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use c_particles
```

  ⟨*Standard module head*⟩

  ⟨*Lorentz: public*⟩

  ⟨*Lorentz: public operators*⟩

  ⟨*Lorentz: public functions*⟩

  ⟨*Lorentz: types*⟩

  ⟨*Lorentz: parameters*⟩

  ⟨*Lorentz: interfaces*⟩

```
  contains
```

  ⟨*Lorentz: procedures*⟩
```
  end module lorentz
```

4.2.1 Three-vectors

First of all, let us introduce three-vectors in a trivial way. The functions and overloaded elementary operations clearly are too much overhead, but we like to keep the interface for three-vectors and four-vectors exactly parallel. By the way, we might attach a label to a vector by extending the type definition later.

⟨*Lorentz: public*⟩≡
```
    public :: vector3_t
```

⟨*Lorentz: types*⟩≡
```
    type :: vector3_t
       private
       real(default), dimension(3) :: p
    end type vector3_t
```

Output a vector

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_write
```

⟨*Lorentz: procedures*⟩≡
```
    subroutine vector3_write (p, unit, testflag)
      type(vector3_t), intent(in) :: p
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      character(len=7) :: fmt
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call pac_fmt (fmt, FMT_19, FMT_15, testflag)
      write(u, "(1x,A,3(1x," // fmt // "))") 'P = ', p%p
    end subroutine vector3_write
```

This is a three-vector with zero components

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_null
```

⟨*Lorentz: parameters*⟩≡
```
    type(vector3_t), parameter :: vector3_null = &
         vector3_t ([ 0._default, 0._default, 0._default ])
```

Canonical three-vector:

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_canonical
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_canonical (k) result (p)
      type(vector3_t) :: p
      integer, intent(in) :: k
      p = vector3_null
      p%p(k) = 1
    end function vector3_canonical
```

A moving particle (k-axis, or arbitrary axis). Note that the function for the generic momentum cannot be elemental.

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_moving
```

⟨*Lorentz: interfaces*⟩≡
```
    interface vector3_moving
       module procedure vector3_moving_canonical
       module procedure vector3_moving_generic
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_moving_canonical (p, k) result(q)
      type(vector3_t) :: q
      real(default), intent(in) :: p
      integer, intent(in) :: k
      q = vector3_null
      q%p(k) = p
```

237

```
      end function vector3_moving_canonical
      pure function vector3_moving_generic (p) result(q)
        real(default), dimension(3), intent(in) :: p
        type(vector3_t) :: q
        q%p = p
      end function vector3_moving_generic
```

Equality and inequality

⟨*Lorentz: public operators*⟩≡
```
      public :: operator(==), operator(/=)
```

⟨*Lorentz: interfaces*⟩+≡
```
      interface operator(==)
         module procedure vector3_eq
      end interface
      interface operator(/=)
         module procedure vector3_neq
      end interface
```

⟨*Lorentz: procedures*⟩+≡
```
      elemental function vector3_eq (p, q) result (r)
        logical :: r
        type(vector3_t), intent(in) :: p,q
        r = all (p%p == q%p)
      end function vector3_eq
      elemental function vector3_neq (p, q) result (r)
        logical :: r
        type(vector3_t), intent(in) :: p,q
        r = any (p%p /= q%p)
      end function vector3_neq
```

Define addition and subtraction

⟨*Lorentz: public operators*⟩+≡
```
      public :: operator(+), operator(-)
```

⟨*Lorentz: interfaces*⟩+≡
```
      interface operator(+)
         module procedure add_vector3
      end interface
      interface operator(-)
         module procedure sub_vector3
      end interface
```

⟨*Lorentz: procedures*⟩+≡
```
      elemental function add_vector3 (p, q) result (r)
        type(vector3_t) :: r
        type(vector3_t), intent(in) :: p,q
        r%p = p%p + q%p
      end function add_vector3
      elemental function sub_vector3 (p, q) result (r)
        type(vector3_t) :: r
        type(vector3_t), intent(in) :: p,q
        r%p = p%p - q%p
      end function sub_vector3
```

The multiplication sign is overloaded with scalar multiplication; similarly division:

⟨*Lorentz: public operators*⟩+≡
```
    public :: operator(*), operator(/)
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(*)
       module procedure prod_integer_vector3, prod_vector3_integer
       module procedure prod_real_vector3, prod_vector3_real
    end interface
    interface operator(/)
       module procedure div_vector3_real, div_vector3_integer
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function prod_real_vector3 (s, p) result (q)
      type(vector3_t) :: q
      real(default), intent(in) :: s
      type(vector3_t), intent(in) :: p
      q%p = s * p%p
    end function prod_real_vector3
    elemental function prod_vector3_real (p, s) result (q)
      type(vector3_t) :: q
      real(default), intent(in) :: s
      type(vector3_t), intent(in) :: p
      q%p = s * p%p
    end function prod_vector3_real
    elemental function div_vector3_real (p, s) result (q)
      type(vector3_t) :: q
      real(default), intent(in) :: s
      type(vector3_t), intent(in) :: p
      q%p = p%p/s
    end function div_vector3_real
    elemental function prod_integer_vector3 (s, p) result (q)
      type(vector3_t) :: q
      integer, intent(in) :: s
      type(vector3_t), intent(in) :: p
      q%p = s * p%p
    end function prod_integer_vector3
    elemental function prod_vector3_integer (p, s) result (q)
      type(vector3_t) :: q
      integer, intent(in) :: s
      type(vector3_t), intent(in) :: p
      q%p = s * p%p
    end function prod_vector3_integer
    elemental function div_vector3_integer (p, s) result (q)
      type(vector3_t) :: q
      integer, intent(in) :: s
      type(vector3_t), intent(in) :: p
      q%p = p%p/s
    end function div_vector3_integer
```

The multiplication sign can also indicate scalar products:

⟨*Lorentz: interfaces*⟩+≡

```
    interface operator(*)
       module procedure prod_vector3
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function prod_vector3 (p, q) result (s)
      real(default) :: s
      type(vector3_t), intent(in) :: p,q
      s = dot_product (p%p, q%p)
    end function prod_vector3
```

⟨*Lorentz: public functions*⟩≡
```
    public :: cross_product
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface cross_product
       module procedure vector3_cross_product
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_cross_product (p, q) result (r)
      type(vector3_t) :: r
      type(vector3_t), intent(in) :: p,q
      integer :: i
      do i=1,3
         r%p(i) = dot_product (p%p, matmul(epsilon_three(i,:,:), q%p))
      end do
    end function vector3_cross_product
```

Exponentiation is defined only for integer powers. Odd powers mean take the square root; so p**1 is the length of p.

⟨*Lorentz: public operators*⟩+≡
```
    public :: operator(**)
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(**)
       module procedure power_vector3
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function power_vector3 (p, e) result (s)
      real(default) :: s
      type(vector3_t), intent(in) :: p
      integer, intent(in) :: e
      s = dot_product (p%p, p%p)
      if (e/=2) then
         if (mod(e,2)==0) then
            s = s**(e/2)
         else
            s = sqrt(s)**e
         end if
      end if
    end function power_vector3
```

Finally, we need a negation.

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(-)
       module procedure negate_vector3
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function negate_vector3 (p) result (q)
      type(vector3_t) :: q
      type(vector3_t), intent(in) :: p
      integer :: i
      do i = 1, 3
         if (p%p(i) == -p%p(i)) then
            q%p(i) = 0
         else
            q%p(i) = -p%p(i)
         end if
      end do
    end function negate_vector3
```

The sum function can be useful:

⟨*Lorentz: public functions*⟩+≡
```
    public :: sum
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface sum
       module procedure sum_vector3
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    pure function sum_vector3 (p) result (q)
      type(vector3_t) :: q
      type(vector3_t), dimension(:), intent(in) :: p
      integer :: i
      do i=1, 3
         q%p(i) = sum (p%p(i))
      end do
    end function sum_vector3
```

Any component:

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_get_component
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_get_component (p, k) result (c)
      type(vector3_t), intent(in) :: p
      integer, intent(in) :: k
      real(default) :: c
      c = p%p(k)
    end function vector3_get_component
```

Extract all components. This is not elemental.

⟨*Lorentz: public*⟩+≡
```
    public :: vector3_get_components
```

⟨*Lorentz: procedures*⟩+≡
```
    pure function vector3_get_components (p) result (a)
      type(vector3_t), intent(in) :: p
      real(default), dimension(3) :: a
      a = p%p
    end function vector3_get_components
```

This function returns the direction of a three-vector, i.e., a normalized three-vector. If the vector is null, we return a null vector.

⟨*Lorentz: public functions*⟩+≡
```
    public :: direction
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface direction
       module procedure vector3_get_direction
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_get_direction (p) result (q)
      type(vector3_t) :: q
      type(vector3_t), intent(in) :: p
      real(default) :: pp
      pp = p**1
      if (pp /= 0) then
         q%p = p%p / pp
      else
         q%p = 0
      end if
    end function vector3_get_direction
```

4.2.2 Four-vectors

In four-vectors the zero-component needs special treatment, therefore we do not use the standard operations. Sure, we pay for the extra layer of abstraction by losing efficiency; so we have to assume that the time-critical applications do not involve four-vector operations.

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_t
```

⟨*Lorentz: types*⟩+≡
```
    type :: vector4_t
      private
      real(default), dimension(0:3) :: p = &
         [0._default, 0._default, 0._default, 0._default]
    end type vector4_t
```

Output a vector

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_write
```

⟨*Lorentz: procedures*⟩+≡
```
    subroutine vector4_write (p, unit, show_mass, testflag)
      type(vector4_t), intent(in) :: p
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: show_mass
      logical, intent(in), optional :: testflag
      integer :: u
      character(len=7) :: fmt
      call pac_fmt (fmt, FMT_19, FMT_15, testflag)
      u = output_unit (unit);  if (u < 0)  return
      write(u, "(1x,A,1x," // fmt // ")") 'E = ', p%p(0)
      write(u, "(1x,A,3(1x," // fmt // "))") 'P = ', p%p(1:)
      if (present (show_mass)) then
         if (show_mass) &
              write (u, "(1x,A,1x," // fmt // ")") 'M = ', p**1
      end if
    end subroutine vector4_write
```

Binary I/O

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_write_raw
    public :: vector4_read_raw
```

⟨*Lorentz: procedures*⟩+≡
```
    subroutine vector4_write_raw (p, u)
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: u
      write (u) p%p
    end subroutine vector4_write_raw

    subroutine vector4_read_raw (p, u, iostat)
      type(vector4_t), intent(out) :: p
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) p%p
    end subroutine vector4_read_raw
```

This is a four-vector with zero components

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_null
```

⟨*Lorentz: parameters*⟩+≡
```
    type(vector4_t), parameter :: vector4_null = &
         vector4_t ([ 0._default, 0._default, 0._default, 0._default ])
```

Canonical four-vector:

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_canonical
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector4_canonical (k) result (p)
      type(vector4_t) :: p
      integer, intent(in) :: k
      p = vector4_null
```

243

```
      p%p(k) = 1
    end function vector4_canonical
```

A particle at rest:

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_at_rest
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector4_at_rest (m) result (p)
      type(vector4_t) :: p
      real(default), intent(in) :: m
      p = vector4_t ([ m, 0._default, 0._default, 0._default ])
    end function vector4_at_rest
```

A moving particle (k-axis, or arbitrary axis)

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_moving
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface vector4_moving
       module procedure vector4_moving_canonical
       module procedure vector4_moving_generic
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector4_moving_canonical (E, p, k) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: E, p
      integer, intent(in) :: k
      q = vector4_at_rest(E)
      q%p(k) = p
    end function vector4_moving_canonical
    elemental function vector4_moving_generic (E, p) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: E
      type(vector3_t), intent(in) :: p
      q%p(0) = E
      q%p(1:) = p%p
    end function vector4_moving_generic
```

Equality and inequality

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(==)
       module procedure vector4_eq
    end interface
    interface operator(/=)
       module procedure vector4_neq
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector4_eq (p, q) result (r)
      logical :: r
      type(vector4_t), intent(in) :: p,q
      r = all (p%p == q%p)
```

```
    end function vector4_eq
    elemental function vector4_neq (p, q) result (r)
      logical :: r
      type(vector4_t), intent(in) :: p,q
      r = any (p%p /= q%p)
    end function vector4_neq
```

Addition and subtraction:

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(+)
       module procedure add_vector4
    end interface
    interface operator(-)
       module procedure sub_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function add_vector4 (p,q) result (r)
      type(vector4_t) :: r
      type(vector4_t), intent(in) :: p,q
      r%p = p%p + q%p
    end function add_vector4
    elemental function sub_vector4 (p,q) result (r)
      type(vector4_t) :: r
      type(vector4_t), intent(in) :: p,q
      r%p = p%p - q%p
    end function sub_vector4
```

We also need scalar multiplication and division:

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(*)
       module procedure prod_real_vector4, prod_vector4_real
       module procedure prod_integer_vector4, prod_vector4_integer
    end interface
    interface operator(/)
       module procedure div_vector4_real
       module procedure div_vector4_integer
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function prod_real_vector4 (s, p) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = s * p%p
    end function prod_real_vector4
    elemental function prod_vector4_real (p, s) result (q)
      type(vector4_t) :: q
      real(default), intent(in) :: s
      type(vector4_t), intent(in) :: p
      q%p = s * p%p
    end function prod_vector4_real
    elemental function div_vector4_real (p, s) result (q)
      type(vector4_t) :: q
```

```
    real(default), intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = p%p/s
  end function div_vector4_real
  elemental function prod_integer_vector4 (s, p) result (q)
    type(vector4_t) :: q
    integer, intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = s * p%p
  end function prod_integer_vector4
  elemental function prod_vector4_integer (p, s) result (q)
    type(vector4_t) :: q
    integer, intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = s * p%p
  end function prod_vector4_integer
  elemental function div_vector4_integer (p, s) result (q)
    type(vector4_t) :: q
    integer, intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = p%p/s
  end function div_vector4_integer
```

Scalar products and squares in the Minkowski sense:

⟨*Lorentz: interfaces*⟩+≡

```
  interface operator(*)
    module procedure prod_vector4
  end interface
  interface operator(**)
    module procedure power_vector4
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function prod_vector4 (p, q) result (s)
    real(default) :: s
    type(vector4_t), intent(in) :: p,q
    s = p%p(0)*q%p(0) - dot_product(p%p(1:), q%p(1:))
  end function prod_vector4
```

The power operation for four-vectors is signed, i.e., `p**1` is positive for timelike and negative for spacelike vectors. Note that `(p**1)**2` is not necessarily equal to `p**2`.

⟨*Lorentz: procedures*⟩+≡

```
  elemental function power_vector4 (p, e) result (s)
    real(default) :: s
    type(vector4_t), intent(in) :: p
    integer, intent(in) :: e
    s = p*p
    if (e/=2) then
       if (mod(e,2)==0) then
          s = s**(e/2)
       elseif (s>=0) then
          s = sqrt(s)**e
```

```
        else
            s = -(sqrt(abs(s))**e)
        end if
    end if
end function power_vector4
```

Finally, we introduce a negation

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(-)
       module procedure negate_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function negate_vector4 (p) result (q)
      type(vector4_t) :: q
      type(vector4_t), intent(in) :: p
      integer :: i
      do i = 0, 3
         if (p%p(i) == -p%p(i)) then
            q%p(i) = 0
         else
            q%p(i) = -p%p(i)
         end if
      end do
    end function negate_vector4
```

The sum function can be useful:

⟨*Lorentz: interfaces*⟩+≡
```
    interface sum
       module procedure sum_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    pure function sum_vector4 (p) result (q)
      type(vector4_t) :: q
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: i
      do i=0, 3
         q%p(i) = sum (p%p(i))
      end do
    end function sum_vector4
```

4.2.3 Conversions

Manually set a component of the four-vector:

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_set_component
```

⟨*Lorentz: procedures*⟩+≡
```
    subroutine vector4_set_component (p, k, c)
      type(vector4_t), intent(inout) :: p
      integer, intent(in) :: k
      real(default), intent(in) :: c
```

```
      p%p(k) = c
    end subroutine vector4_set_component
```

Any component:

⟨Lorentz: public⟩+≡
```
    public :: vector4_get_component
```

⟨Lorentz: procedures⟩+≡
```
    elemental function vector4_get_component (p, k) result (c)
      real(default) :: c
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: k
      c = p%p(k)
    end function vector4_get_component
```

Extract all components. This is not elemental.

⟨Lorentz: public⟩+≡
```
    public :: vector4_get_components
```

⟨Lorentz: procedures⟩+≡
```
    pure function vector4_get_components (p) result (a)
      real(default), dimension(0:3) :: a
      type(vector4_t), intent(in) :: p
      a = p%p
    end function vector4_get_components
```

This function returns the space part of a four-vector, such that we can apply three-vector operations on it:

⟨Lorentz: public functions⟩+≡
```
    public :: space_part
```

⟨Lorentz: interfaces⟩+≡
```
    interface space_part
       module procedure vector4_get_space_part
    end interface
```

⟨Lorentz: procedures⟩+≡
```
    elemental function vector4_get_space_part (p) result (q)
      type(vector3_t) :: q
      type(vector4_t), intent(in) :: p
      q%p = p%p(1:)
    end function vector4_get_space_part
```

This function returns the direction of a four-vector, i.e., a normalized three-vector. If the four-vector has zero space part, we return a null vector.

⟨Lorentz: interfaces⟩+≡
```
    interface direction
       module procedure vector4_get_direction
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function vector4_get_direction (p) result (q)
  type(vector3_t) :: q
  type(vector4_t), intent(in) :: p
  real(default) :: qq
  q%p = p%p(1:)
  qq = q**1
  if (qq /= 0) then
     q%p = q%p / qq
  else
     q%p = 0
  end if
end function vector4_get_direction
```

This function returns the four-vector as an ordinary array. A second version for an array of four-vectors.

⟨*Lorentz: public functions*⟩+≡
```
public :: array_from_vector4
```

⟨*Lorentz: interfaces*⟩+≡
```
interface array_from_vector4
   module procedure array_from_vector4_1
   module procedure array_from_vector4_2
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
pure function array_from_vector4_1 (p) result (a)
  type(vector4_t), intent(in) :: p
  real(default), dimension(0:3) :: a
  a = p%p
end function array_from_vector4_1

pure function array_from_vector4_2 (p) result (a)
  type(vector4_t), dimension(:), intent(in) :: p
  real(default), dimension(0:3, size(p)) :: a
  integer :: i
  forall (i=1:size(p))
     a(0:3,i) = p(i)%p
  end forall
end function array_from_vector4_2
```

Transform the momentum of a `c_prt` object into a four-vector:

⟨*Lorentz: public*⟩+≡
```
public :: vector4_from_c_prt
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function vector4_from_c_prt (c_prt) result (p)
  type(vector4_t) :: p
  type(c_prt_t), intent(in) :: c_prt
  p%p(0) = c_prt%pe
  p%p(1) = c_prt%px
  p%p(2) = c_prt%py
  p%p(3) = c_prt%pz
end function vector4_from_c_prt
```

Initialize a `c_prt_t` object with the components of a four-vector as its kinematical entries. Compute the invariant mass, or use the optional mass-squared value instead.

⟨*Lorentz: public*⟩+≡
```
    public :: vector4_to_c_prt
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector4_to_c_prt (p, p2) result (c_prt)
      type(c_prt_t) :: c_prt
      type(vector4_t), intent(in) :: p
      real(default), intent(in), optional :: p2
      c_prt%pe = p%p(0)
      c_prt%px = p%p(1)
      c_prt%py = p%p(2)
      c_prt%pz = p%p(3)
      if (present (p2)) then
         c_prt%p2 = p2
      else
         c_prt%p2 = p ** 2
      end if
    end function vector4_to_c_prt
```

4.2.4 Angles

Return the angles in a canonical system. The angle ϕ is defined between $0 \leq \phi < 2\pi$. In degenerate cases, return zero.

⟨*Lorentz: public functions*⟩+≡
```
    public :: azimuthal_angle
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface azimuthal_angle
       module procedure vector3_azimuthal_angle
       module procedure vector4_azimuthal_angle
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_azimuthal_angle (p) result (phi)
      real(default) :: phi
      type(vector3_t), intent(in) :: p
      if (any(p%p(1:2)/=0)) then
         phi = atan2(p%p(2), p%p(1))
         if (phi < 0) phi = phi + twopi
      else
         phi = 0
      end if
    end function vector3_azimuthal_angle
    elemental function vector4_azimuthal_angle (p) result (phi)
      real(default) :: phi
      type(vector4_t), intent(in) :: p
      phi = vector3_azimuthal_angle (space_part (p))
    end function vector4_azimuthal_angle
```

Azimuthal angle in degrees

⟨*Lorentz: public functions*⟩+≡
```
    public :: azimuthal_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface azimuthal_angle_deg
       module procedure vector3_azimuthal_angle_deg
       module procedure vector4_azimuthal_angle_deg
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_azimuthal_angle_deg (p) result (phi)
      real(default) :: phi
      type(vector3_t), intent(in) :: p
      phi = vector3_azimuthal_angle (p) / degree
    end function vector3_azimuthal_angle_deg
    elemental function vector4_azimuthal_angle_deg (p) result (phi)
      real(default) :: phi
      type(vector4_t), intent(in) :: p
      phi = vector4_azimuthal_angle (p) / degree
    end function vector4_azimuthal_angle_deg
```

The azimuthal distance of two vectors. This is the difference of the azimuthal angles, but cannot be larger than π: The result is between $-\pi < \Delta\phi \leq \pi$.

⟨*Lorentz: public functions*⟩+≡
```
    public :: azimuthal_distance
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface azimuthal_distance
       module procedure vector3_azimuthal_distance
       module procedure vector4_azimuthal_distance
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_azimuthal_distance (p, q) result (dphi)
      real(default) :: dphi
      type(vector3_t), intent(in) :: p,q
      dphi = vector3_azimuthal_angle (q) - vector3_azimuthal_angle (p)
      if (dphi <= -pi) then
         dphi = dphi + twopi
      else if (dphi > pi) then
         dphi = dphi - twopi
      end if
    end function vector3_azimuthal_distance
    elemental function vector4_azimuthal_distance (p, q) result (dphi)
      real(default) :: dphi
      type(vector4_t), intent(in) :: p,q
      dphi = vector3_azimuthal_distance &
           (space_part (p), space_part (q))
    end function vector4_azimuthal_distance
```

The same in degrees:

⟨*Lorentz: public functions*⟩+≡
```
    public :: azimuthal_distance_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface azimuthal_distance_deg
       module procedure vector3_azimuthal_distance_deg
       module procedure vector4_azimuthal_distance_deg
    end interface
```
⟨*Lorentz: procedures*⟩+≡
```
    elemental function vector3_azimuthal_distance_deg (p, q) result (dphi)
      real(default) :: dphi
      type(vector3_t), intent(in) :: p,q
      dphi = vector3_azimuthal_distance (p, q) / degree
    end function vector3_azimuthal_distance_deg
    elemental function vector4_azimuthal_distance_deg (p, q) result (dphi)
      real(default) :: dphi
      type(vector4_t), intent(in) :: p,q
      dphi = vector4_azimuthal_distance (p, q) / degree
    end function vector4_azimuthal_distance_deg
```

The polar angle is defined $0 \leq \theta \leq \pi$. Note that `ATAN2` has the reversed order of arguments: `ATAN2(Y,X)`. Here, x is the 3-component while y is the transverse momentum which is always nonnegative. Therefore, the result is nonnegative as well.

⟨*Lorentz: public functions*⟩+≡
```
    public :: polar_angle
```
⟨*Lorentz: interfaces*⟩+≡
```
    interface polar_angle
       module procedure polar_angle_vector3
       module procedure polar_angle_vector4
    end interface
```
⟨*Lorentz: procedures*⟩+≡
```
    elemental function polar_angle_vector3 (p) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p
      if (any(p%p/=0)) then
         theta = atan2 (sqrt(p%p(1)**2 + p%p(2)**2), p%p(3))
      else
         theta = 0
      end if
    end function polar_angle_vector3
    elemental function polar_angle_vector4 (p) result (theta)
      real(default) :: theta
      type(vector4_t), intent(in) :: p
      theta = polar_angle (space_part (p))
    end function polar_angle_vector4
```

This is the cosine of the polar angle: $-1 \leq \cos\theta \leq 1$.

⟨*Lorentz: public functions*⟩+≡
```
    public :: polar_angle_ct
```
⟨*Lorentz: interfaces*⟩+≡
```
    interface polar_angle_ct
       module procedure polar_angle_ct_vector3
       module procedure polar_angle_ct_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function polar_angle_ct_vector3 (p) result (ct)
      real(default) :: ct
      type(vector3_t), intent(in) :: p
      if (any(p%p/=0)) then
         ct = p%p(3) / p**1
      else
         ct = 1
      end if
    end function polar_angle_ct_vector3
    elemental function polar_angle_ct_vector4 (p) result (ct)
      real(default) :: ct
      type(vector4_t), intent(in) :: p
      ct = polar_angle_ct (space_part (p))
    end function polar_angle_ct_vector4
```

The polar angle in degrees.

⟨*Lorentz: public functions*⟩+≡
```
    public :: polar_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface polar_angle_deg
       module procedure polar_angle_deg_vector3
       module procedure polar_angle_deg_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function polar_angle_deg_vector3 (p) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p
      theta = polar_angle (p) / degree
    end function polar_angle_deg_vector3
    elemental function polar_angle_deg_vector4 (p) result (theta)
      real(default) :: theta
      type(vector4_t), intent(in) :: p
      theta = polar_angle (p) / degree
    end function polar_angle_deg_vector4
```

This is the angle enclosed between two three-momenta. If one of the momenta is zero, we return an angle of zero. The range of the result is $0 \leq \theta \leq \pi$. If there is only one argument, take the positive z axis as reference.

⟨*Lorentz: public functions*⟩+≡
```
    public :: enclosed_angle
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface enclosed_angle
       module procedure enclosed_angle_vector3
       module procedure enclosed_angle_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function enclosed_angle_vector3 (p, q) result (theta)
      real(default) :: theta
      type(vector3_t), intent(in) :: p, q
      theta = acos (enclosed_angle_ct (p, q))
```

```
      end function enclosed_angle_vector3
      elemental function enclosed_angle_vector4 (p, q) result (theta)
        real(default) :: theta
        type(vector4_t), intent(in) :: p, q
        theta = enclosed_angle (space_part (p), space_part (q))
      end function enclosed_angle_vector4
```

The cosine of the enclosed angle.

⟨Lorentz: public functions⟩+≡
```
      public :: enclosed_angle_ct
```

⟨Lorentz: interfaces⟩+≡
```
      interface enclosed_angle_ct
         module procedure enclosed_angle_ct_vector3
         module procedure enclosed_angle_ct_vector4
      end interface
```

⟨Lorentz: procedures⟩+≡
```
      elemental function enclosed_angle_ct_vector3 (p, q) result (ct)
        real(default) :: ct
        type(vector3_t), intent(in) :: p, q
        if (any(p%p/=0).and.any(q%p/=0)) then
           ct = p*q / (p**1 * q**1)
           if (ct>1) then
              ct = 1
           else if (ct<-1) then
              ct = -1
           end if
        else
           ct = 1
        end if
      end function enclosed_angle_ct_vector3
      elemental function enclosed_angle_ct_vector4 (p, q) result (ct)
        real(default) :: ct
        type(vector4_t), intent(in) :: p, q
        ct = enclosed_angle_ct (space_part (p), space_part (q))
      end function enclosed_angle_ct_vector4
```

The enclosed angle in degrees.

⟨Lorentz: public functions⟩+≡
```
      public :: enclosed_angle_deg
```

⟨Lorentz: interfaces⟩+≡
```
      interface enclosed_angle_deg
         module procedure enclosed_angle_deg_vector3
         module procedure enclosed_angle_deg_vector4
      end interface
```

⟨Lorentz: procedures⟩+≡
```
      elemental function enclosed_angle_deg_vector3 (p, q) result (theta)
        real(default) :: theta
        type(vector3_t), intent(in) :: p, q
        theta = enclosed_angle (p, q) / degree
      end function enclosed_angle_deg_vector3
      elemental function enclosed_angle_deg_vector4 (p, q) result (theta)
```

```
      real(default) :: theta
      type(vector4_t), intent(in) :: p, q
      theta = enclosed_angle (p, q) / degree
    end function enclosed_angle_deg_vector4
```

The polar angle of the first momentum w.r.t. the second momentum, evaluated in the rest frame of the second momentum. If the second four-momentum is not timelike, return zero.

⟨*Lorentz: public functions*⟩+≡
```
    public :: enclosed_angle_rest_frame
    public :: enclosed_angle_ct_rest_frame
    public :: enclosed_angle_deg_rest_frame
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface enclosed_angle_rest_frame
       module procedure enclosed_angle_rest_frame_vector4
    end interface
    interface enclosed_angle_ct_rest_frame
       module procedure enclosed_angle_ct_rest_frame_vector4
    end interface
    interface enclosed_angle_deg_rest_frame
       module procedure enclosed_angle_deg_rest_frame_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function enclosed_angle_rest_frame_vector4 (p, q) result (theta)
      type(vector4_t), intent(in) :: p, q
      real(default) :: theta
      theta = acos (enclosed_angle_ct_rest_frame (p, q))
    end function enclosed_angle_rest_frame_vector4
    elemental function enclosed_angle_ct_rest_frame_vector4 (p, q) result (ct)
      type(vector4_t), intent(in) :: p, q
      real(default) :: ct
      if (invariant_mass(q) > 0) then
         ct = enclosed_angle_ct ( &
              space_part (boost(-q, invariant_mass (q)) * p), &
              space_part (q))
      else
         ct = 1
      end if
    end function enclosed_angle_ct_rest_frame_vector4
    elemental function enclosed_angle_deg_rest_frame_vector4 (p, q) &
         result (theta)
      type(vector4_t), intent(in) :: p, q
      real(default) :: theta
      theta = enclosed_angle_rest_frame (p, q) / degree
    end function enclosed_angle_deg_rest_frame_vector4
```

4.2.5   More kinematical functions (some redundant)

The scalar transverse momentum (assuming the z axis is longitudinal)

⟨*Lorentz: public functions*⟩+≡
```
    public :: transverse_part
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface transverse_part
       module procedure transverse_part_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function transverse_part_vector4 (p) result (pT)
      real(default) :: pT
      type(vector4_t), intent(in) :: p
      pT = sqrt(p%p(1)**2 + p%p(2)**2)
    end function transverse_part_vector4
```

The scalar longitudinal momentum (assuming the z axis is longitudinal). Identical to `momentum_z_component`.

⟨*Lorentz: public functions*⟩+≡
```
    public :: longitudinal_part
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface longitudinal_part
       module procedure longitudinal_part_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function longitudinal_part_vector4 (p) result (pL)
      real(default) :: pL
      type(vector4_t), intent(in) :: p
      pL = p%p(3)
    end function longitudinal_part_vector4
```

Absolute value of three-momentum

⟨*Lorentz: public functions*⟩+≡
```
    public :: space_part_norm
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface space_part_norm
       module procedure space_part_norm_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function space_part_norm_vector4 (p) result (p3)
      real(default) :: p3
      type(vector4_t), intent(in) :: p
      p3 = sqrt (p%p(1)**2 + p%p(2)**2 + p%p(3)**2)
    end function space_part_norm_vector4
```

The energy (the zeroth component)

⟨*Lorentz: public functions*⟩+≡
```
    public :: energy
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface energy
       module procedure energy_vector4
       module procedure energy_vector3
       module procedure energy_real
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function energy_vector4 (p) result (E)
  real(default) :: E
  type(vector4_t), intent(in) :: p
  E = p%p(0)
end function energy_vector4
```

Alternative: The energy corresponding to a given momentum and mass. If the mass is omitted, it is zero

⟨*Lorentz: procedures*⟩+≡
```
elemental function energy_vector3 (p, mass) result (E)
  real(default) :: E
  type(vector3_t), intent(in) :: p
  real(default), intent(in), optional :: mass
  if (present (mass)) then
     E = sqrt (p**2 + mass**2)
  else
     E = p**1
  end if
end function energy_vector3

elemental function energy_real (p, mass) result (E)
  real(default) :: E
  real(default), intent(in) :: p
  real(default), intent(in), optional :: mass
  if (present (mass)) then
     E = sqrt (p**2 + mass**2)
  else
     E = abs (p)
  end if
end function energy_real
```

The invariant mass of four-momenta. Zero for lightlike, negative for spacelike momenta.

⟨*Lorentz: public functions*⟩+≡
```
public :: invariant_mass
```

⟨*Lorentz: interfaces*⟩+≡
```
interface invariant_mass
   module procedure invariant_mass_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function invariant_mass_vector4 (p) result (m)
  real(default) :: m
  type(vector4_t), intent(in) :: p
  real(default) :: msq
  msq = p*p
  if (msq >= 0) then
     m = sqrt (msq)
  else
     m = - sqrt (abs (msq))
  end if
end function invariant_mass_vector4
```

257

The invariant mass squared. Zero for lightlike, negative for spacelike momenta.

⟨*Lorentz: public functions*⟩+≡
```
    public :: invariant_mass_squared
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface invariant_mass_squared
       module procedure invariant_mass_squared_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function invariant_mass_squared_vector4 (p) result (msq)
      real(default) :: msq
      type(vector4_t), intent(in) :: p
      msq = p*p
    end function invariant_mass_squared_vector4
```

The transverse mass. If the mass squared is negative, this value also is negative.

⟨*Lorentz: public functions*⟩+≡
```
    public :: transverse_mass
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface transverse_mass
       module procedure transverse_mass_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function transverse_mass_vector4 (p) result (m)
      real(default) :: m
      type(vector4_t), intent(in) :: p
      real(default) :: msq
      msq = p%p(0)**2 - p%p(1)**2 - p%p(2)**2
      if (msq >= 0) then
         m = sqrt (msq)
      else
         m = - sqrt (abs (msq))
      end if
    end function transverse_mass_vector4
```

The rapidity (defined if particle is massive or $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡
```
    public :: rapidity
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface rapidity
       module procedure rapidity_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function rapidity_vector4 (p) result (y)
      real(default) :: y
      type(vector4_t), intent(in) :: p
      y = .5 * log( (energy (p) + longitudinal_part (p)) &
           &        /(energy (p) - longitudinal_part (p)))
    end function rapidity_vector4
```

The pseudorapidity (defined if $p_\perp > 0$)

⟨Lorentz: public functions⟩+≡
```
    public :: pseudorapidity
```

⟨Lorentz: interfaces⟩+≡
```
    interface pseudorapidity
       module procedure pseudorapidity_vector4
    end interface
```

⟨Lorentz: procedures⟩+≡
```
    elemental function pseudorapidity_vector4 (p) result (eta)
      real(default) :: eta
      type(vector4_t), intent(in) :: p
      eta = -log( tan (.5 * polar_angle (p)))
    end function pseudorapidity_vector4
```

The rapidity distance (defined if both $p_\perp > 0$)

⟨Lorentz: public functions⟩+≡
```
    public :: rapidity_distance
```

⟨Lorentz: interfaces⟩+≡
```
    interface rapidity_distance
       module procedure rapidity_distance_vector4
    end interface
```

⟨Lorentz: procedures⟩+≡
```
    elemental function rapidity_distance_vector4 (p, q) result (dy)
      type(vector4_t), intent(in) :: p, q
      real(default) :: dy
      dy = rapidity (q) - rapidity (p)
    end function rapidity_distance_vector4
```

The pseudorapidity distance (defined if both $p_\perp > 0$)

⟨Lorentz: public functions⟩+≡
```
    public :: pseudorapidity_distance
```

⟨Lorentz: interfaces⟩+≡
```
    interface pseudorapidity_distance
       module procedure pseudorapidity_distance_vector4
    end interface
```

⟨Lorentz: procedures⟩+≡
```
    elemental function pseudorapidity_distance_vector4 (p, q) result (deta)
      real(default) :: deta
      type(vector4_t), intent(in) :: p, q
      deta = pseudorapidity (q) - pseudorapidity (p)
    end function pseudorapidity_distance_vector4
```

The distance on the $\eta - \phi$ cylinder:

⟨Lorentz: public functions⟩+≡
```
    public :: eta_phi_distance
```

⟨Lorentz: interfaces⟩+≡
```
    interface eta_phi_distance
       module procedure eta_phi_distance_vector4
    end interface
```

⟨*Lorentz: procedures*⟩+≡

```
    elemental function eta_phi_distance_vector4 (p, q) result (dr)
      type(vector4_t), intent(in) :: p, q
      real(default) :: dr
      dr = sqrt ( &
           pseudorapidity_distance (p, q)**2 &
           + azimuthal_distance (p, q)**2)
    end function eta_phi_distance_vector4
```

4.2.6   Lorentz transformations

⟨*Lorentz: public*⟩+≡

```
    public :: lorentz_transformation_t
```

⟨*Lorentz: types*⟩+≡

```
    type :: lorentz_transformation_t
      private
      real(default), dimension(0:3, 0:3) :: L
     contains
     ⟨Lorentz: lorentz transformation: TBP⟩
    end type lorentz_transformation_t
```

Output:

⟨*Lorentz: public*⟩+≡

```
    public :: lorentz_transformation_write
```

⟨*Lorentz: lorentz transformation: TBP*⟩≡

```
    procedure :: write => lorentz_transformation_write
```

⟨*Lorentz: procedures*⟩+≡

```
    subroutine lorentz_transformation_write (L, unit)
      class(lorentz_transformation_t), intent(in) :: L
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A,3(1x," // FMT_19 // "))")  "L00 = ", L%L(0,0)
      write (u, "(1x,A,3(1x," // FMT_19 // "))")  "L0j = ", L%L(0,1:3)
      do i = 1, 3
         write (u, "(1x,A,I0,A,3(1x," // FMT_19 // "))")  &
              "L", i, "0 = ", L%L(i,0)
         write (u, "(1x,A,I0,A,3(1x," // FMT_19 // "))")  &
              "L", i, "j = ", L%L(i,1:3)
      end do
    end subroutine lorentz_transformation_write
```

Extract all components:

⟨*Lorentz: public*⟩+≡

```
    public :: lorentz_transformation_get_components
```

⟨*Lorentz: procedures*⟩+≡

```
    pure function lorentz_transformation_get_components (L) result (a)
      type(lorentz_transformation_t), intent(in) :: L
```

```
      real(default), dimension(0:3,0:3) :: a
      a = L%L
    end function lorentz_transformation_get_components
```

4.2.7   Functions of Lorentz transformations

For the inverse, we make use of the fact that $\Lambda^{\mu\nu}\Lambda_{\mu\rho} = \delta^\nu_\rho$. So, lowering the indices and transposing is sufficient.

⟨*Lorentz: public functions*⟩+≡
```
    public :: inverse
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface inverse
       module procedure lorentz_transformation_inverse
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function lorentz_transformation_inverse (L) result (IL)
      type(lorentz_transformation_t) :: IL
      type(lorentz_transformation_t), intent(in) :: L
      IL%L(0,0) = L%L(0,0)
      IL%L(0,1:) = -L%L(1:,0)
      IL%L(1:,0) = -L%L(0,1:)
      IL%L(1:,1:) = transpose(L%L(1:,1:))
    end function lorentz_transformation_inverse
```

4.2.8   Invariants

These are used below. The first array index is varying fastest in `FORTRAN`; therefore the extra minus in the odd-rank tensor epsilon.

⟨*Lorentz: parameters*⟩+≡
```
    integer, dimension(3,3), parameter :: delta_three = &
        & reshape( source = (/ 1,0,0, 0,1,0, 0,0,1 /), &
        &          shape  = (/3,3/) )
    integer, dimension(3,3,3), parameter :: epsilon_three = &
        & reshape( source = (/ 0, 0,0,  0,0,-1,   0,1,0,&
        &                      0, 0,1,  0,0, 0,  -1,0,0,&
        &                      0,-1,0,  1,0, 0,   0,0,0 /),&
        &          shape = (/3,3,3/) )
```

This could be of some use:

⟨*Lorentz: public*⟩+≡
```
    public :: identity
```

⟨*Lorentz: parameters*⟩+≡
```
    type(lorentz_transformation_t), parameter :: &
        & identity = &
        & lorentz_transformation_t ( &
        & reshape( source = (/ 1._default, 0._default, 0._default, 0._default, &
        &                      0._default, 1._default, 0._default, 0._default, &
        &                      0._default, 0._default, 1._default, 0._default, &
        &                      0._default, 0._default, 0._default, 1._default /),&
        &          shape = (/ 4,4 /) ) )
```

⟨*Lorentz: public*⟩+≡
```
    public :: space_reflection
```

⟨*Lorentz: parameters*⟩+≡
```
    type(lorentz_transformation_t), parameter :: &
        & space_reflection = &
        & lorentz_transformation_t ( &
        & reshape( source = (/ 1._default, 0._default, 0._default, 0._default, &
        &                      0._default,-1._default, 0._default, 0._default, &
        &                      0._default, 0._default,-1._default, 0._default, &
        &                      0._default, 0._default, 0._default,-1._default /),&
        &            shape = (/ 4,4 /) ) )
```

4.2.9 Boosts

We build Lorentz transformations from boosts and rotations. In both cases we can supply a three-vector which defines the axis and (hyperbolic) angle. For a boost, this is the vector $\vec{\beta} = \vec{p}/E$, such that a particle at rest with mass m is boosted to a particle with three-vector $\vec{p}$. Here, we have

$$\beta = \tanh \chi = p/E, \qquad \gamma = \cosh \chi = E/m, \qquad \beta\gamma = \sinh \chi = p/m \qquad (4.1)$$

⟨*Lorentz: public functions*⟩+≡
```
    public :: boost
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface boost
       module procedure boost_from_rest_frame
       module procedure boost_from_rest_frame_vector3
       module procedure boost_generic
       module procedure boost_canonical
    end interface
```

In the first form, the argument is some four-momentum, the space part of which determines a direction, and the associated mass (which is not checked against the four-momentum). The boost vector $\gamma\vec{\beta}$ is then given by $\vec{p}/m$. This boosts from the rest frame of a particle to the current frame. To be explicit, if $\vec{p}$ is the momentum of a particle and m its mass, $L(\vec{p}/m)$ is the transformation that turns $(m; \vec{0})$ into $(E; \vec{p})$. Conversely, the inverse transformation boosts a vector *into* the rest frame of a particle, in particular $(E; \vec{p})$ into $(m; \vec{0})$.

⟨*Lorentz: procedures*⟩+≡
```
    elemental function boost_from_rest_frame (p, m) result (L)
      type(lorentz_transformation_t) :: L
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: m
      L = boost_from_rest_frame_vector3 (space_part (p), m)
    end function boost_from_rest_frame
    elemental function boost_from_rest_frame_vector3 (p, m) result (L)
      type(lorentz_transformation_t) :: L
      type(vector3_t), intent(in) :: p
      real(default), intent(in) :: m
      type(vector3_t) :: beta_gamma
      real(default) :: bg2, g, c
      integer :: i,j
```

```
    if (m /= 0) then
       beta_gamma = p / m
       bg2 = beta_gamma**2
    else
       bg2 = 0
    end if
    if (bg2 /= 0) then
       g = sqrt(1 + bg2);  c = (g-1)/bg2
       L%L(0,0)  = g
       L%L(0,1:) = beta_gamma%p
       L%L(1:,0) = L%L(0,1:)
       do i=1,3
          do j=1,3
             L%L(i,j) = delta_three(i,j) + c*beta_gamma%p(i)*beta_gamma%p(j)
          end do
       end do
    else
       L = identity
    end if
    end function boost_from_rest_frame_vector3
```

A canonical boost is a boost along one of the coordinate axes, which we may supply as an integer argument. Here, $\gamma\beta$ is scalar.

⟨*Lorentz: procedures*⟩+≡

```
    elemental function boost_canonical (beta_gamma, k) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: beta_gamma
      integer, intent(in) :: k
      real(default) :: g
      g = sqrt(1 + beta_gamma**2)
      L = identity
      L%L(0,0) = g
      L%L(0,k) = beta_gamma
      L%L(k,0) = L%L(0,k)
      L%L(k,k) = L%L(0,0)
    end function boost_canonical
```

Instead of a canonical axis, we can supply an arbitrary axis which need not be normalized. If it is zero, return the unit matrix.

⟨*Lorentz: procedures*⟩+≡

```
    elemental function boost_generic (beta_gamma, axis) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: beta_gamma
      type(vector3_t), intent(in) :: axis
      if (any(axis%p/=0)) then
         L = boost_from_rest_frame_vector3 (beta_gamma * axis, axis**1)
      else
         L = identity
      end if
    end function boost_generic
```

4.2.10 Rotations

For a rotation, the vector defines the rotation axis, and its length the rotation angle.

⟨*Lorentz: public functions*⟩+≡
```
    public :: rotation
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface rotation
       module procedure rotation_generic
       module procedure rotation_canonical
       module procedure rotation_generic_cs
       module procedure rotation_canonical_cs
    end interface
```

If $\cos\phi$ and $\sin\phi$ is already known, we do not have to calculate them. Of course, the user has to ensure that $\cos^2\phi + \sin^2\phi = 1$, and that the given axis n is normalized to one. In the second form, the length of `axis` is the rotation angle.

⟨*Lorentz: procedures*⟩+≡
```
    elemental function rotation_generic_cs (cp, sp, axis) result (R)
      type(lorentz_transformation_t) :: R
      real(default), intent(in) :: cp, sp
      type(vector3_t), intent(in) :: axis
      integer :: i,j
      R = identity
      do i=1,3
         do j=1,3
            R%L(i,j) = cp*delta_three(i,j) + (1-cp)*axis%p(i)*axis%p(j)  &
                 &    - sp*dot_product(epsilon_three(i,j,:), axis%p)
         end do
      end do
    end function rotation_generic_cs
    elemental function rotation_generic (axis) result (R)
      type(lorentz_transformation_t) :: R
      type(vector3_t), intent(in) :: axis
      real(default) :: phi
      if (any(axis%p/=0)) then
         phi = abs(axis**1)
         R = rotation_generic_cs (cos(phi), sin(phi), axis/phi)
      else
         R = identity
      end if
    end function rotation_generic
```

Alternatively, give just the angle and label the coordinate axis by an integer.

⟨*Lorentz: procedures*⟩+≡
```
    elemental function rotation_canonical_cs (cp, sp, k) result (R)
      type(lorentz_transformation_t) :: R
      real(default), intent(in) :: cp, sp
      integer, intent(in) :: k
      integer :: i,j
      R = identity
      do i=1,3
         do j=1,3
```

```
            R%L(i,j) = -sp*epsilon_three(i,j,k)
          end do
          R%L(i,i) = cp
        end do
        R%L(k,k) = 1
      end function rotation_canonical_cs
      elemental function rotation_canonical (phi, k) result (R)
        type(lorentz_transformation_t) :: R
        real(default), intent(in) :: phi
        integer, intent(in) :: k
        R = rotation_canonical_cs(cos(phi), sin(phi), k)
      end function rotation_canonical
```

This is viewed as a method for the first argument (three-vector): Reconstruct the rotation that rotates it into the second three-vector.

⟨*Lorentz: public functions*⟩+≡
```
    public :: rotation_to_2nd
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface rotation_to_2nd
       module procedure rotation_to_2nd_generic
       module procedure rotation_to_2nd_canonical
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function rotation_to_2nd_generic (p, q) result (R)
      type(lorentz_transformation_t) :: R
      type(vector3_t), intent(in) :: p, q
      type(vector3_t) :: a, b, ab
      real(default) :: ct, st
      if (any (p%p /= 0) .and. any (q%p /= 0)) then
         a = direction (p)
         b = direction (q)
         ab = cross_product(a,b)
         ct = a*b;   st = ab**1
         if (st /= 0) then
            R = rotation_generic_cs (ct, st, ab/st)
         else if (ct < 0) then
            R = space_reflection
         else
            R = identity
         end if
      else
         R = identity
      end if
    end function rotation_to_2nd_generic
```

The same for a canonical axis: The function returns the transformation that rotates the k-axis into the direction of p.

⟨*Lorentz: procedures*⟩+≡
```
    elemental function rotation_to_2nd_canonical (k, p) result (R)
      type(lorentz_transformation_t) :: R
      integer, intent(in) :: k
      type(vector3_t), intent(in) :: p
      type(vector3_t) :: b, ab
      real(default) :: ct, st
```

```
    integer :: i, j
    if (any (p%p /= 0)) then
       b = direction (p)
       ab%p = 0
       do i = 1, 3
          do j = 1, 3
             ab%p(j) = ab%p(j) + b%p(i) * epsilon_three(i,j,k)
          end do
       end do
       ct = b%p(k);  st = ab**1
       if (st /= 0) then
          R = rotation_generic_cs (ct, st, ab/st)
       else if (ct < 0) then
          R = space_reflection
       else
          R = identity
       end if
    else
       R = identity
    end if
end function rotation_to_2nd_canonical
```

4.2.11 Composite Lorentz transformations

This function returns the transformation that, given a pair of vectors $p_{1,2}$, (a) boosts from the rest frame of the c.m. system (with invariant mass m) into the lab frame where p_i are defined, and (b) turns the given axis (or the canonical vectors $\pm e_k$) in the rest frame into the directions of $p_{1,2}$ in the lab frame. Note that the energy components are not used; for a consistent result one should have $(p_1 + p_2)^2 = m^2$.

⟨*Lorentz: public functions*⟩+≡
```
    public :: transformation
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface transformation
       module procedure transformation_rec_generic
       module procedure transformation_rec_canonical
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function transformation_rec_generic (axis, p1, p2, m) result (L)
      type(vector3_t), intent(in) :: axis
      type(vector4_t), intent(in) :: p1, p2
      real(default), intent(in) :: m
      type(lorentz_transformation_t) :: L
      L = boost (p1 + p2, m)
      L = L * rotation_to_2nd (axis, space_part (inverse (L) * p1))
    end function transformation_rec_generic
    elemental function transformation_rec_canonical (k, p1, p2, m) result (L)
      integer, intent(in) :: k
      type(vector4_t), intent(in) :: p1, p2
      real(default), intent(in) :: m
      type(lorentz_transformation_t) :: L
```

```
    L = boost (p1 + p2, m)
    L = L * rotation_to_2nd (k, space_part (inverse (L) * p1))
  end function transformation_rec_canonical
```

4.2.12 Applying Lorentz transformations

Multiplying vectors and Lorentz transformations is straightforward.

⟨*Lorentz: interfaces*⟩+≡
```
    interface operator(*)
       module procedure prod_LT_vector4
       module procedure prod_LT_LT
       module procedure prod_vector4_LT
    end interface
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function prod_LT_vector4 (L, p) result (np)
      type(vector4_t) :: np
      type(lorentz_transformation_t), intent(in) :: L
      type(vector4_t), intent(in) :: p
      np%p = matmul (L%L, p%p)
    end function prod_LT_vector4
    elemental function prod_LT_LT (L1, L2) result (NL)
      type(lorentz_transformation_t) :: NL
      type(lorentz_transformation_t), intent(in) :: L1,L2
      NL%L = matmul (L1%L, L2%L)
    end function prod_LT_LT
    elemental function prod_vector4_LT (p, L) result (np)
      type(vector4_t) :: np
      type(vector4_t), intent(in) :: p
      type(lorentz_transformation_t), intent(in) :: L
      np%p = matmul (p%p, L%L)
    end function prod_vector4_LT
```

4.2.13 Special Lorentz transformations

These routines have their application in the generation and extraction of angles in the phase-space sampling routine. Since this part of the program is time-critical, we calculate the composition of transformations directly instead of multiplying rotations and boosts.

This Lorentz transformation is the composition of a rotation by ϕ around the 3 axis, a rotation by θ around the 2 axis, and a boost along the 3 axis:

$$L = B_3(\beta\gamma) \, R_2(\theta) \, R_3(\phi) \tag{4.2}$$

Instead of the angles we provide sine and cosine.

⟨*Lorentz: public functions*⟩+≡
```
    public :: LT_compose_r3_r2_b3
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function LT_compose_r3_r2_b3 &
        (cp, sp, ct, st, beta_gamma) result (L)
      type(lorentz_transformation_t) :: L
```

```
      real(default), intent(in) :: cp, sp, ct, st, beta_gamma
      real(default) :: gamma
      if (beta_gamma==0) then
         L%L(0,0)  = 1
         L%L(1:,0) = 0
         L%L(0,1:) = 0
         L%L(1,1:) = (/  ct*cp, -ct*sp, st /)
         L%L(2,1:) = (/     sp,     cp, 0._default /)
         L%L(3,1:) = (/ -st*cp,  st*sp, ct /)
      else
         gamma = sqrt(1 + beta_gamma**2)
         L%L(0,0)  = gamma
         L%L(1,0)  = 0
         L%L(2,0)  = 0
         L%L(3,0)  = beta_gamma
         L%L(0,1:) = beta_gamma * (/ -st*cp,  st*sp, ct /)
         L%L(1,1:) =                 (/  ct*cp, -ct*sp, st /)
         L%L(2,1:) =                 (/     sp,     cp, 0._default /)
         L%L(3,1:) = gamma       * (/ -st*cp,  st*sp, ct /)
      end if
    end function LT_compose_r3_r2_b3
```

Different ordering:
$$L = B_3(\beta\gamma)\, R_3(\phi)\, R_2(\theta) \tag{4.3}$$

⟨*Lorentz: public functions*⟩+≡
```
    public :: LT_compose_r2_r3_b3
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function LT_compose_r2_r3_b3 &
        (ct, st, cp, sp, beta_gamma) result (L)
      type(lorentz_transformation_t) :: L
      real(default), intent(in) :: ct, st, cp, sp, beta_gamma
      real(default) :: gamma
      if (beta_gamma==0) then
         L%L(0,0)  = 1
         L%L(1:,0) = 0
         L%L(0,1:) = 0
         L%L(1,1:) = (/  ct*cp,    -sp,     st*cp /)
         L%L(2,1:) = (/  ct*sp,     cp,     st*sp /)
         L%L(3,1:) = (/ -st  , 0._default, ct    /)
      else
         gamma = sqrt(1 + beta_gamma**2)
         L%L(0,0)  = gamma
         L%L(1,0)  = 0
         L%L(2,0)  = 0
         L%L(3,0)  = beta_gamma
         L%L(0,1:) = beta_gamma * (/ -st  , 0._default, ct    /)
         L%L(1,1:) =                 (/  ct*cp,    -sp,     st*cp /)
         L%L(2,1:) =                 (/  ct*sp,     cp,     st*sp /)
         L%L(3,1:) = gamma       * (/ -st  , 0._default, ct    /)
      end if
    end function LT_compose_r2_r3_b3
```

This function returns the previous Lorentz transformation applied to an arbitrary four-momentum and extracts the space part of the result:

$$\vec{n} = [B_3(\beta\gamma)\, R_2(\theta)\, R_3(\phi)\, p]_{\text{space part}} \tag{4.4}$$

The second variant applies if there is no rotation

⟨*Lorentz: public functions*⟩+≡
```
    public :: axis_from_p_r3_r2_b3, axis_from_p_b3
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function axis_from_p_r3_r2_b3 &
        (p, cp, sp, ct, st, beta_gamma) result (n)
      type(vector3_t) :: n
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: cp, sp, ct, st, beta_gamma
      real(default) :: gamma, px, py
      px = cp * p%p(1) - sp * p%p(2)
      py = sp * p%p(1) + cp * p%p(2)
      n%p(1) =  ct * px + st * p%p(3)
      n%p(2) = py
      n%p(3) = -st * px + ct * p%p(3)
      if (beta_gamma/=0) then
         gamma = sqrt(1 + beta_gamma**2)
         n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
      end if
    end function axis_from_p_r3_r2_b3

    elemental function axis_from_p_b3 (p, beta_gamma) result (n)
      type(vector3_t) :: n
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: beta_gamma
      real(default) :: gamma
      n%p = p%p(1:3)
      if (beta_gamma/=0) then
         gamma = sqrt(1 + beta_gamma**2)
         n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
      end if
    end function axis_from_p_b3
```

4.2.14   Special functions

The standard phase space function:

⟨*Lorentz: public functions*⟩+≡
```
    public :: lambda
```

⟨*Lorentz: procedures*⟩+≡
```
    elemental function lambda (m1sq, m2sq, m3sq)
      real(default) :: lambda
      real(default), intent(in) :: m1sq, m2sq, m3sq
      lambda = (m1sq - m2sq - m3sq)**2 - 4*m2sq*m3sq
    end function lambda
```

Return a pair of head-to-head colliding momenta, given the collider energy, particle masses, and optionally the momentum of the c.m. system.

⟨*Lorentz: public functions*⟩+≡
```
    public :: colliding_momenta
```

⟨*Lorentz: procedures*⟩+≡
```
    function colliding_momenta (sqrts, m, p_cm) result (p)
      type(vector4_t), dimension(2) :: p
      real(default), intent(in) :: sqrts
      real(default), dimension(2), intent(in), optional :: m
      real(default), intent(in), optional :: p_cm
      real(default), dimension(2) :: dmsq
      real(default) :: ch, sh
      real(default), dimension(2) :: E0, p0
      integer, dimension(2), parameter :: sgn = (/1, -1/)
      if (sqrts == 0) then
         call msg_fatal (" Colliding beams: sqrts is zero (please set sqrts)")
         p = vector4_null;  return
      else if (sqrts <= 0) then
         call msg_fatal (" Colliding beams: sqrts is negative")
         p = vector4_null;  return
      end if
      if (present (m)) then
         dmsq = sgn * (m(1)**2-m(2)**2)
         E0 = (sqrts + dmsq/sqrts) / 2
         if (any (E0 < m)) then
            call msg_fatal &
                 (" Colliding beams: beam energy is less than particle mass")
            p = vector4_null;  return
         end if
         p0 = sgn * sqrt (E0**2 - m**2)
      else
         E0 = sqrts / 2
         p0 = sgn * E0
      end if
      if (present (p_cm)) then
         sh = p_cm / sqrts
         ch = sqrt (1 + sh**2)
         p = vector4_moving (E0 * ch + p0 * sh, E0 * sh + p0 * ch, 3)
      else
         p = vector4_moving (E0, p0, 3)
      end if
    end function colliding_momenta
```

This subroutine is for the purpose of numerical checks and comparisons. The idea is to set a number to zero if it is numerically equivalent with zero. The equivalence is established by comparing with a `tolerance` argument. We implement this for vectors and transformations.

⟨*Lorentz: public functions*⟩+≡
```
    public :: pacify
```

⟨*Lorentz: interfaces*⟩+≡
```
    interface pacify
       module procedure pacify_vector3
```

```
      module procedure pacify_vector4
      module procedure pacify_LT
    end interface pacify
```

⟨*Lorentz: procedures*⟩+≡

```
    elemental subroutine pacify_vector3 (p, tolerance)
      type(vector3_t), intent(inout) :: p
      real(default), intent(in) :: tolerance
      where (abs (p%p) < tolerance)  p%p = 0
    end subroutine pacify_vector3

    elemental subroutine pacify_vector4 (p, tolerance)
      type(vector4_t), intent(inout) :: p
      real(default), intent(in) :: tolerance
      where (abs (p%p) < tolerance)  p%p = 0
    end subroutine pacify_vector4

    elemental subroutine pacify_LT (LT, tolerance)
      type(lorentz_transformation_t), intent(inout) :: LT
      real(default), intent(in) :: tolerance
      where (abs (LT%L) < tolerance)  LT%L = 0
    end subroutine pacify_LT
```

4.3 Special Physics functions

Here, we declare functions that are specific for the Standard Model, including QCD: fixed and running α_s, Catani-Seymour dipole terms, loop functions, etc.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

⟨sm_physics.f90⟩≡
  ⟨*File header*⟩

  ```
  module sm_physics
  ```

  ⟨*Use kinds*⟩
  ```
    use constants !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use limits, only: MZ_REF, ALPHA_QCD_MZ_REF, LAMBDA_QCD_REF !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
  ```

  ⟨*Standard module head*⟩

  ⟨*SM physics: public*⟩

  ⟨*SM physics: parameters*⟩

  ```
  contains
  ```

  ⟨*SM physics: procedures*⟩

  ```
  end module sm_physics
  ```

4.3.1 Running α_s

First we set a reference value for $\alpha_s(M_Z) = 0.1178$.

⟨*Limits: public parameters*⟩+≡
  ```
    real(default), public, parameter :: MZ_REF = 91.188_default
    real(default), public, parameter :: ALPHA_QCD_MZ_REF = 0.1178_default
    real(default), public, parameter :: LAMBDA_QCD_REF = 200.e-3_default
  ```

Then we define the coefficients of the beta function of QCD (as a reference cf. the Particle Data Group), where n_f is the number of active flavors in two different schemes:

$$\beta_0 = 11 - \frac{2}{3}n_f \qquad (4.5)$$

$$\beta_1 = 51 - \frac{19}{3}n_f \qquad (4.6)$$

$$\beta_2 = 2857 - \frac{5033}{9}n_f + \frac{325}{27}n_f^2 \qquad (4.7)$$

$$b_0 = \frac{1}{12\pi} \left(11C_A - 2n_f\right) \tag{4.8}$$

$$b_1 = \frac{1}{24\pi^2} \left(17C_A^2 - 5C_A n_f - 3C_F n_f\right) \tag{4.9}$$

$$b_2 = \frac{1}{(4\pi)^3} \left(\frac{2857}{54}C_A^3 - \frac{1415}{54} * C_A^2 n_f - \frac{205}{18}C_A C_F n_f + C_F^2 n_f + \frac{79}{54}C_A n_f**2 + \frac{11}{9}C_F n_f**2\right) \tag{4.10}$$

⟨*SM physics: public*⟩≡
```
public :: beta0, beta1, beta2, coeff_b0, coeff_b1, coeff_b2
```

⟨*SM physics: procedures*⟩≡
```
pure function beta0 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta0
  beta0 = 11.0_default - two/three * nf
end function beta0

pure function beta1 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta1
  beta1 = 51.0_default - 19.0_default/three * nf
end function beta1

pure function beta2 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta2
  beta2 = 2857.0_default - 5033.0_default / 9.0_default * &
                 nf + 325.0_default/27.0_default * nf**2
end function beta2

pure function coeff_b0 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b0
  coeff_b0 = (11.0_default * CA - two * nf) / (12.0_default * pi)
end function coeff_b0

pure function coeff_b1 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b1
  coeff_b1 = (17.0_default * CA**2 - five * CA * nf - three * CF * nf) / &
            (24.0_default * pi**2)
end function coeff_b1

pure function coeff_b2 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b2
  coeff_b2 = (2857.0_default/54.0_default * CA**3 - &
                 1415.0_default/54.0_default * &
                 CA**2 * nf - 205.0_default/18.0_default * CA*CF*nf &
                 + 79.0_default/54.0_default * CA*nf**2 + &
                 11.0_default/9.0_default * CF * nf**2) / (four*pi)**3
end function coeff_b2
```

There should be two versions of running α_s, one which takes the scale and Λ_{QCD} as imput, and one which takes the scale and e.g. $\alpha_s(m_Z)$ as input. Here, we take the one which takes the QCD scale and scale as inputs from the PDG book.

⟨*SM physics: public*⟩+≡
```
    public :: running_as, running_as_lam
```

⟨*SM physics: procedures*⟩+≡
```
    pure function running_as (scale, al_mz, mz, order, nf) result (ascale)
      real(default), intent(in) :: scale
      real(default), intent(in), optional :: al_mz, nf, mz
      integer, intent(in), optional :: order
      integer :: ord
      real(default) :: az, m_z, as_log, n_f, b0, b1, b2, ascale
      real(default) :: as0, as1
      if (present(mz)) then
        m_z = mz
      else
        m_z = MZ_REF
      end if
      if (present(order)) then
         ord = order
      else
         ord = 0
      end if
      if (present(al_mz)) then
         az = al_mz
      else
         az = ALPHA_QCD_MZ_REF
      end if
      if (present(nf)) then
        n_f = nf
      else
        n_f = 5
      end if
      b0 = coeff_b0 (n_f)
      b1 = coeff_b1 (n_f)
      b2 = coeff_b2 (n_f)
      as_log = one + b0 * az * log(scale**2/m_z**2)
      as0 = az / as_log
      as1 = as0 - as0**2 * b1/b0 * log(as_log)
      select case (ord)
        case (0)
          ascale = as0
        case (1)
          ascale = as1
        case (2)
          ascale = as1 + as0**3 * (b1**2/b0**2 * ((log(as_log))**2 - &
                     log(as_log) + as_log - one) - b2/b0 * (as_log - one))
      end select
    end function running_as

    pure function running_as_lam (nf, scale, lambda, order) result (ascale)
      real(default), intent(in) :: nf, scale
      real(default), intent(in), optional :: lambda
```

```
      integer, intent(in), optional :: order
      real(default) :: lambda_qcd
      real(default) :: as0, as1, logmul, b0, b1, b2, ascale
      integer :: ord
      if (present (lambda)) then
         lambda_qcd = lambda
      else
         lambda_qcd = LAMBDA_QCD_REF
      end if
      if (present(order)) then
        ord = order
      else
        ord = 0
      end if
      b0 = beta0(nf)
      b1 = beta1(nf)
      b2 = beta2(nf)
      logmul = log(scale**2/lambda_qcd**2)
      as0 = four*pi / b0 / logmul
      as1 = as0 * (one - two* b1 / b0**2 * log(logmul) / logmul)
      select case (ord)
        case (0)
          ascale = as0
        case (1)
          ascale = as1
        case (2)
          ascale = as1 + as0 * four * b1**2/b0**4/logmul**2 * &
                 ( (log(logmul) - 0.5_default)**2 + &
                 b2*b0/8.0_default/b1**2 - five/four)
      end select
    end function running_as_lam
```

4.3.2 Catani-Seymour Parameters

These are fundamental constants of the Catani-Seymour dipole formalism. Since the corresponding parameters for the gluon case depend on the number of flavors which is treated as an argument, there we do have functions and not parameters.

$$\gamma_q = \gamma_{\bar{q}} = \frac{3}{2}C_F \qquad \gamma_g = \frac{11}{6}C_A - \frac{2}{3}T_R N_f \qquad (4.11)$$

$$K_q = K_{\bar{q}} = \left(\frac{7}{2} - \frac{\pi^2}{6}\right)C_F \qquad K_g = \left(\frac{67}{18} - \frac{\pi^2}{6}\right)C_A - \frac{10}{9}T_R N_f \qquad (4.12)$$

⟨*SM physics: parameters*⟩≡
```
    real(kind=default), parameter, public ::  gamma_q = three/two * CF, &
      k_q = (7.0_default/two - pi**2/6.0_default) * CF
```

⟨*SM physics: public*⟩+≡
```
    public :: gamma_g, k_g
```

⟨*SM physics: procedures*⟩+≡
```
elemental function gamma_g (nf) result (gg)
   real(kind=default), intent(in) :: nf
   real(kind=default) :: gg
   gg = 11.0_default/6.0_default * CA - two/three * TR * nf
end function gamma_g

elemental function k_g (nf) result (kg)
   real(kind=default), intent(in) :: nf
   real(kind=default) :: kg
   kg = (67.0_default/18.0_default - pi**2/6.0_default) * CA - &
             10.0_default/9.0_default * TR * nf
end function k_g
```

4.3.3 Mathematical Functions

The dilogarithm. This simplified version is bound to double precision, and restricted to argument values less or equal to unity, so we do not need complex algebra. The wrapper converts it to default precision (which is, of course, a no-op if double=default).

The routine calculates the dilogarithm through mapping on the area where there is a quickly convergent series (adapted from an F77 routine by Hans Kuijf, 1988): Map x such that x is not in the neighbourhood of 1. Note that $|z| = -\ln(1-x)$ is always smaller than 1.10, but $\frac{1 \cdot 10^{19}}{19!}$Bernoulli$_{19} = 2.7 \times 10^{-15}$.

⟨*SM physics: public*⟩+≡
```
public :: Li2
```

⟨*SM physics: procedures*⟩+≡
```
function Li2 (x)
    use kinds, only: double !NODEP!
    real(default), intent(in) :: x
    real(default) :: Li2
    Li2 = real( Li2_double (real(x, kind=double)), kind=default)
end function Li2
```

⟨*SM physics: procedures*⟩+≡
```
function Li2_double (x)  result (Li2)
  use kinds, only: double !NODEP!
  real(kind=double), intent(in) :: x
  real(kind=double) :: Li2
  real(kind=double), parameter :: pi2_6 = pi**2/6
  if (abs(1-x) < 1.E-13_double) then
     Li2 = pi2_6
  else if (abs(1-x) <  0.5_double) then
     Li2 = pi2_6 - log(1-x) * log(x) - Li2_restricted (1-x)
  else if (abs(x).gt.1.d0) then
     call msg_bug (" Dilogarithm called outside of defined range.")
     !!! Insert dilogarithm identity, not used yet
     ! Li2 = -pi2_6 - 0.5_default * log(-x) * log(-x) - Li2_restricted (1/x)
  else
     Li2 = Li2_restricted (x)
```

276

```
          end if
       contains
         function Li2_restricted (x) result (Li2)
           real(kind=double), intent(in) :: x
           real(kind=double) :: Li2
           real(kind=double) :: tmp, z, z2
           z = - log (1-x)
           z2 = z**2
   ! Horner's rule for the powers z^3 through z^19
           tmp = 43867._double/798._double
           tmp = tmp * z2 /342._double - 3617._double/510._double
           tmp = tmp * z2 /272._double + 7._double/6._double
           tmp = tmp * z2 /210._double - 691._double/2730._double
           tmp = tmp * z2 /156._double + 5._double/66._double
           tmp = tmp * z2 /110._double - 1._double/30._double
           tmp = tmp * z2 / 72._double + 1._double/42._double
           tmp = tmp * z2 / 42._double - 1._double/30._double
           tmp = tmp * z2 / 20._double + 1._double/6._double
   ! The first three terms of the power series
           Li2 = z2 * z * tmp / 6._double - 0.25_double * z2 + z
         end function Li2_restricted
       end function Li2_double
```

4.3.4 Loop Integrals

These functions appear in the calculation of the effective one-loop coupling of a (pseudo)scalar to a vector boson pair.

⟨*SM physics: public*⟩+≡
```
       public :: faux
```

⟨*SM physics: procedures*⟩+≡
```
       elemental function faux (x) result (y)
         real(default), intent(in) :: x
         complex(default) :: y
         if (1 <= x) then
            y = asin(sqrt(1/x))**2
         else
            y = - 1/4.0_default * (log((1 + sqrt(1 - x))/ &
                 (1 - sqrt(1 - x))) - cmplx (0.0_default, pi, kind=default))**2
         end if
       end function faux
```

⟨*SM physics: public*⟩+≡
```
       public :: fonehalf
```

⟨*SM physics: procedures*⟩+≡
```
     elemental function fonehalf (x) result (y)
       real(default), intent(in) :: x
       complex(default) :: y
       if (x==0) then
          y = 0
       else
          y = - 2.0_default * x * (1 + (1 - x) * faux(x))
```

```
         end if
      end function fonehalf
```

⟨*SM physics: public*⟩+≡
```
      public :: fonehalf_pseudo
```

⟨*SM physics: procedures*⟩+≡
```
      function fonehalf_pseudo (x) result (y)
        real(default), intent(in) :: x
        complex(default) :: y
        if (x==0) then
           y = 0
        else
           y = - 2.0_default * x * faux(x)
        end if
      end function fonehalf_pseudo
```

⟨*SM physics: public*⟩+≡
```
      public :: fone
```

⟨*SM physics: procedures*⟩+≡
```
      elemental function fone (x) result  (y)
        real(default), intent(in) :: x
        complex(default) :: y
        if (x==0) then
           y = 2.0_default
        else
           y = 2.0_default + 3.0_default * x + &
                 3.0_default * x * (2.0_default - x) * &
                 faux(x)
        end if
      end function fone
```

⟨*SM physics: public*⟩+≡
```
      public :: gaux
```

⟨*SM physics: procedures*⟩+≡
```
      elemental function gaux (x) result (y)
        real(default), intent(in) :: x
        complex(default) :: y
        if (1 <= x) then
           y = sqrt(x - 1) * asin(sqrt(1/x))
        else
           y = sqrt(1 - x) * (log((1 + sqrt(1 - x)) / &
                 (1 - sqrt(1 - x))) - &
                 cmplx (0.0_default, pi, kind=default)) / 2.0_default
        end if
      end function gaux
```

⟨*SM physics: public*⟩+≡
```
      public :: tri_i1
```

⟨*SM physics: procedures*⟩+≡

```
elemental function tri_i1 (a,b) result (y)
  real(default), intent(in) :: a,b
  complex(default) :: y
  if (a < epsilon(a) .or. b < epsilon (b)) then
     y = 0
  else
     y = a*b/2.0_default/(a-b) + a**2 * b**2/2.0_default/(a-b)**2 * &
           (faux(a) - faux(b)) + &
           a**2 * b/(a-b)**2 * (gaux(a) - gaux(b))
  end if
end function tri_i1
```

⟨*SM physics: public*⟩+≡

```
public :: tri_i2
```

⟨*SM physics: procedures*⟩+≡

```
elemental function tri_i2 (a,b) result (y)
  real(default), intent(in) :: a,b
  complex(default) :: y
  if (a < epsilon (a) .or. b < epsilon(b)) then
     y = 0
  else
     y = - a * b / 2.0_default / (a-b) * (faux(a) - faux(b))
  end if
end function tri_i2
```

4.3.5 More on α_s

These functions are for the running of the strong coupling constants, α_s.

⟨*SM physics: public*⟩+≡

```
public :: run_b0
```

⟨*SM physics: procedures*⟩+≡

```
elemental function run_b0 (nf) result (bnull)
  integer, intent(in) :: nf
  real(default) :: bnull
  bnull = 33.0_default - 2.0_default * nf
end function run_b0
```

⟨*SM physics: public*⟩+≡

```
public :: run_b1
```

⟨*SM physics: procedures*⟩+≡

```
elemental function run_b1 (nf) result (bone)
    integer, intent(in) :: nf
    real(default) :: bone
    bone = 6.0_default * (153.0_default - 19.0_default * nf)/run_b0(nf)**2
end function run_b1
```

⟨*SM physics: public*⟩+≡

```
public :: run_aa
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function run_aa (nf) result (aaa)
        integer, intent(in) :: nf
        real(default) :: aaa
        aaa = 12.0_default * PI / run_b0(nf)
    end function run_aa
```

⟨*SM physics: pubic functions*⟩≡
```
    public :: run_bb
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function run_bb (nf) result (bbb)
      integer, intent(in) :: nf
      real(default) :: bbb
      bbb = run_b1(nf) / run_aa(nf)
    end function run_bb
```

4.3.6   Functions for Catani-Seymour dipoles

For the automated Catani-Seymour dipole subtraction, we need the following
functions.

⟨*SM physics: public*⟩+≡
```
    public :: ff_dipole
```

⟨*SM physics: procedures*⟩+≡
```
    pure subroutine ff_dipole (v_ijk,y_ijk,p_ij,pp_k,p_i,p_j,p_k)
        type(vector4_t), intent(in) :: p_i, p_j, p_k
        type(vector4_t), intent(out) :: p_ij, pp_k
        real(kind=default), intent(out) :: y_ijk
        real(kind=default) :: z_i
        real(kind=default), intent(out) :: v_ijk
        z_i   = (p_i*p_k) / ((p_k*p_j) + (p_k*p_i))
        y_ijk = (p_i*p_j) / ((p_i*p_j) + (p_i*p_k) + (p_j*p_k))
        p_ij  = p_i + p_j - y_ijk/(1.0_default - y_ijk) * p_k
        pp_k  = (1.0/(1.0_default - y_ijk)) * p_k
        !!! We don't multiply by alpha_s right here:
        v_ijk = 8.0_default * PI * CF * &
            (2.0 / (1.0 - z_i*(1.0 - y_ijk)) - (1.0 + z_i))
    end subroutine ff_dipole
```

⟨*SM physics: public*⟩+≡
```
    public :: fi_dipole
```

⟨*SM physics: procedures*⟩+≡
```
    pure subroutine fi_dipole (v_ija,x_ija,p_ij,pp_a,p_i,p_j,p_a)
      type(vector4_t), intent(in) :: p_i, p_j, p_a
      type(vector4_t), intent(out) :: p_ij, pp_a
      real(kind=default), intent(out) :: x_ija
      real(kind=default) :: z_i
      real(kind=default), intent(out) :: v_ija
      z_i   = (p_i*p_a) / ((p_a*p_j) + (p_a*p_i))
      x_ija = ((p_i*p_a) + (p_j*p_a) - (p_i*p_j)) &
```

```
          / ((p_i*p_a) + (p_j*p_a))
      p_ij  = p_i + p_j - (1.0_default - x_ija) * p_a
      pp_a  = x_ija * p_a
      !!! We don't not multiply by alpha_s right here:
      v_ija = 8.0_default * PI * CF * &
          (2.0 / (1.0 - z_i + (1.0 - x_ija)) - (1.0 + z_i)) / x_ija
    end subroutine fi_dipole
```

⟨*SM physics: public*⟩+≡
```
    public :: if_dipole
```

⟨*SM physics: procedures*⟩+≡
```
    pure subroutine if_dipole (v_kja,u_j,p_aj,pp_k,p_k,p_j,p_a)
      type(vector4_t), intent(in) :: p_k, p_j, p_a
      type(vector4_t), intent(out) :: p_aj, pp_k
      real(kind=default), intent(out) :: u_j
      real(kind=default) :: x_kja
      real(kind=default), intent(out) :: v_kja
      u_j   = (p_a*p_j) / ((p_a*p_j) + (p_a*p_k))
      x_kja = ((p_a*p_k) + (p_a*p_j) - (p_j*p_k)) &
            / ((p_a*p_j) + (p_a*p_k))
      p_aj  = x_kja * p_a
      pp_k  = p_k + p_j - (1.0_default - x_kja) * p_a
      v_kja = 8.0_default * PI * CF * &
            (2.0 / (1.0 - x_kja + u_j) - (1.0 + x_kja)) / x_kja
    end subroutine if_dipole
```

This function depends on a variable number of final state particles whose kinematics all get changed by the initial-initial dipole insertion.

⟨*SM physics: public*⟩+≡
```
    public :: ii_dipole
```

⟨*SM physics: procedures*⟩+≡
```
    pure subroutine ii_dipole (v_jab,v_j,p_in,p_out,flag_1or2)
        type(vector4_t), dimension(:), intent(in) :: p_in
        type(vector4_t), dimension(size(p_in)-1), intent(out) :: p_out
        logical, intent(in) :: flag_1or2
        real(kind=default), intent(out) :: v_j
        real(kind=default), intent(out) :: v_jab
        type(vector4_t) :: p_a, p_b, p_j
        type(vector4_t) :: k, kk
        type(vector4_t) :: p_aj
        real(kind=default) :: x_jab
        integer :: i
        !!! flag_1or2 decides whether this a 12 or 21 dipole
        if (flag_1or2) then
           p_a = p_in(1)
           p_b = p_in(2)
        else
           p_b = p_in(1)
           p_a = p_in(2)
        end if
        !!! We assume that the unresolved particle has always the last
        !!! momentum
```

```
        p_j = p_in(size(p_in))
        x_jab = ((p_a*p_b) - (p_a*p_j) - (p_b*p_j)) / (p_a*p_b)
        v_j = (p_a*p_j) / (p_a * p_b)
        p_aj  = x_jab * p_a
        k     = p_a + p_b - p_j
        kk    = p_aj + p_b
        do i = 3, size(p_in)-1
           p_out(i) = p_in(i) - 2.0*((k+kk)*p_in(i))/((k+kk)*(k+kk)) * (k+kk) + &
                 (2.0 * (k*p_in(i)) / (k*k)) * kk
        end do
        if (flag_1or2) then
           p_out(1) = p_aj
           p_out(2) = p_b
        else
           p_out(1) = p_b
           p_out(2) = p_aj
        end if
        v_jab = 8.0_default * PI * CF * &
              (2.0 / (1.0 - x_jab) - (1.0 + x_jab)) / x_jab
    end subroutine ii_dipole
```

4.3.7 Distributions for integrated dipoles and such

The Dirac delta distribution, modified for Monte-Carlo sampling:

⟨*SM physics: public*⟩+≡
```
    public :: delta
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function delta (x,eps) result (z)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: z
      if (x > 1.0_default - eps) then
         z = 1.0_default/eps
      else
         z = 0
      end if
    end function delta
```

The +-distribution, $P_+(x) = \left(\frac{1}{1-x}\right)_+$, for the regularization of soft-collinear singularities. The constant part for the Monte-Carlo sampling is the integral over the splitting function divided by the weight for the WHIZARD numerical integration over the interval.

⟨*SM physics: public*⟩+≡
```
    public :: plus_distr
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function plus_distr (x,eps) result (plusd)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: plusd
      if (x > (1.0_default - eps)) then
         plusd = log(eps)/eps
      else
         plusd = one/(one-x)
```

```
            end if
         end function plus_distr
```

The splitting function in $D = 4$ dimensions, regularized as $+$-distributions if necessary:

$$P^{qq}(x) = P^{\bar{q}\bar{q}}(x) = C_F \cdot \left(\frac{1 + x^2}{1 - x}\right)_+ \tag{4.13}$$

$$P^{qg}(x) = P^{\bar{q}g}(x) = C_F \cdot \frac{1 + (1 - x)^2}{x} \tag{4.14}$$

$$P^{gq}(x) = P^{g\bar{q}}(x) = T_R \cdot \left[x^2 + (1 - x)^2\right] \tag{4.15}$$

$$P^{gg}(x) = 2C_A\left[\left(\frac{1}{1 - x}\right)_+ + \frac{1 - x}{x} - 1 + x(1 - x)\right] + \delta(1 - x)\left(\frac{11}{6}C_A - \frac{2}{3}N_f T_R\right) \tag{4.16}$$

Since the number of flavors summed over in the gluon splitting function might depend on the physics case under consideration it is implemented as an input variable.

⟨*SM physics: public*⟩+≡
```
    public :: pqq
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function pqq (x,eps) result (pqqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: pqqx
      if (x > (1.0_default - eps)) then
         pqqx = (eps-one)/two + two*log(eps)/eps - three*(eps-one)/eps/two
      else
         pqqx = (one + x**2)/(one-x)
      end if
      pqqx = CF * pqqx
    end function pqq
```

⟨*SM physics: public*⟩+≡
```
    public :: pgq
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function pgq (x) result (pgqx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pgqx
      pgqx = TR * (x**2 + (one - x)**2)
    end function pgq
```

⟨*SM physics: public*⟩+≡
```
    public :: pqg
```

⟨*SM physics: procedures*⟩+≡
```
    elemental function pqg (x) result (pqgx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pqgx
      pqgx = CF * (one + (one - x)**2) / x
    end function pqg
```

```
    public :: pgg
```
```
    elemental function pgg (x, nf, eps) result (pggx)
      real(kind=default), intent(in) :: x, nf, eps
      real(kind=default) :: pggx
      pggx = two * CA * ( plus_distr (x, eps) + (one-x)/x - one + &
                    x*(one-x)) + delta (x, eps)  * gamma_g(nf)
    end function pgg
```

For the *qq* and *gg* cases, there exist "regularized" versions of the splitting functions:

$$P_{\text{reg}}^{qq}(x) = -C_F \cdot (1 + x) \tag{4.17}$$

$$P_{\text{reg}}^{gg}(x) = 2C_A \left[\frac{1-x}{x} - 1 + x(1-x) \right] \tag{4.18}$$

$$\tag{4.19}$$

```
    public :: pqq_reg
```
```
    elemental function pqq_reg (x) result (pqqregx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pqqregx
      pqqregx = - CF * (one + x)
    end function pqq_reg
```

```
    public :: pgg_reg
```
```
    elemental function pgg_reg (x) result (pggregx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pggregx
      pggregx = two * CA * ((one - x)/x - one + x*(one - x))
    end function pgg_reg
```

Here, we collect the expressions needed for integrated Catani-Seymour dipoles, and the so-called flavor kernels. We always distinguish between the "ordinary" Catani-Seymour version, and the one including a phase-space slicing parameter, α.

The standard flavor kernels $\overline{K}^{ab}$ are:

$$\overline{K}^{qg}(x) = \overline{K}^{\bar{q}g}(x) = P^{qg}(x)\log((1-x)/x) + CF \times x \tag{4.20}$$

$$\overline{K}^{gq}(x) = \overline{K}^{g\bar{q}}(x) = P^{gq}(x)\log((1-x)/x) + TR \times 2x(1-x) \tag{4.21}$$

$$\overline{K}^{qq} = C_F \left[\left(\frac{2}{1-x} \log \frac{1-x}{x} \right)_+ - (1+x)\log((1-x)/x) + (1-x) \right] - (5 - \pi^2) \cdot C_F \cdot \delta(1-x) \tag{4.22}$$

$$\overline{K}^{gg} = 2C_A \left[\left(\frac{1}{1-x} \log \frac{1-x}{x} \right)_+ + \left(\frac{1-x}{x} - 1 + x(1-x) \right) \log((1-x)/x) \right] - delta(1-x) \left[(f \right. \tag{4.23}$$

284

⟨*SM physics: public*⟩+≡
```
    public :: kbarqg
```

⟨*SM physics: procedures*⟩+≡
```
    function kbarqg (x) result (kbarqgx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: kbarqgx
      kbarqgx = pqg(x) * log((one-x)/x) + CF * x
    end function kbarqg
```

⟨*SM physics: public*⟩+≡
```
    public :: kbargq
```

⟨*SM physics: procedures*⟩+≡
```
    function kbargq (x) result (kbargqx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: kbargqx
      kbargqx = pgq(x) * log((one-x)/x) + two * TR * x * (one - x)
    end function kbargq
```

⟨*SM physics: public*⟩+≡
```
    public :: kbarqq
```

⟨*SM physics: procedures*⟩+≡
```
    function kbarqq (x,eps) result (kbarqqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: kbarqqx
      kbarqqx = CF*(log_plus_distr(x,eps) - (one+x) * log((one-x)/x) + (one - &
            x) - (five - pi**2) * delta(x,eps))
    end function kbarqq
```

⟨*SM physics: public*⟩+≡
```
    public :: kbargg
```

⟨*SM physics: procedures*⟩+≡
```
    function kbargg (x,eps,nf) result (kbarggx)
      real(kind=default), intent(in) :: x, eps, nf
      real(kind=default) :: kbarggx
      kbarggx = CA * (log_plus_distr(x,eps) + two * ((one-x)/x - one + &
                      x*(one-x) * log((1-x)/x))) - delta(x,eps) * &
                      ((50.0_default/9.0_default - pi**2) * CA - &
                      16.0_default/9.0_default * TR * nf)
    end function kbargg
```

The $\tilde{K}$ are used when two identified hadrons participate:

$$\tilde{K}^{ab}(x) = P^{ab}_{\text{reg}}(x) \cdot \log(1-x) + \delta^{ab} \mathbf{T}_a^2 \left[\left(\frac{2}{1-x} \log(1-x) \right)_+ - \frac{\pi^2}{3} \delta(1-x) \right] \quad (4.24)$$

⟨*SM physics: public*⟩+≡
```
    public :: ktildeqq
```

⟨*SM physics: procedures*⟩+≡
```
function ktildeqq (x,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeqqx
  ktildeqqx = pqq_reg (x) * log(one-x) + CF * ( - log2_plus_distr (x,eps) &
                  - pi**2/three * delta(x,eps))
end function ktildeqq
```

⟨*SM physics: public*⟩+≡
```
public :: ktildeqg
```
⟨*SM physics: procedures*⟩+≡
```
function ktildeqg (x,eps) result (ktildeqgx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeqgx
  ktildeqgx = pqg (x) * log(one-x)
end function ktildeqg
```

⟨*SM physics: public*⟩+≡
```
public :: ktildegq
```
⟨*SM physics: procedures*⟩+≡
```
function ktildegq (x,eps) result (ktildegqx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildegqx
  ktildegqx = pgq (x) * log(one-x)
end function ktildegq
```

⟨*SM physics: public*⟩+≡
```
public :: ktildegg
```
⟨*SM physics: procedures*⟩+≡
```
function ktildegg (x,eps) result (ktildeggx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: ktildeggx
  ktildeggx = pgg_reg (x) * log(one-x) + CA * ( - &
      log2_plus_distr (x,eps) - pi**2/three * delta(x,eps))
end function ktildegg
```

The insertion operator might not be necessary for a GOLEM interface but is demanded by the Les Houches NLO accord. It is a three-dimensional array, where the index always gives the inverse power of the DREG expansion parameter, ϵ.

⟨*SM physics: public*⟩+≡
```
public :: insert_q
```
⟨*SM physics: procedures*⟩+≡
```
pure function insert_q ()
  real(kind=default), dimension(0:2) :: insert_q
  insert_q(0) = gamma_q + k_q - pi**2/three * CF
  insert_q(1) = gamma_q
  insert_q(2) = CF
end function insert_q
```

⟨*SM physics: public*⟩+≡
```
    public :: insert_g
```

⟨*SM physics: procedures*⟩+≡
```
    pure function insert_g (nf)
      real(kind=default), intent(in) :: nf
      real(kind=default), dimension(0:2) :: insert_g
      insert_g(0) = gamma_g (nf) + k_g (nf) - pi**2/three * CA
      insert_g(1) = gamma_g (nf)
      insert_g(2) = CA
    end function insert_g
```

For better convergence, one can exclude regions of phase space with a slicing parameter from the dipole subtraction procedure. First of all, the K functions get modified:

$$K_i(\alpha) = K_i - \mathbf{T}_i^2 \log^2 \alpha + \gamma_i(\alpha - 1 - \log \alpha) \tag{4.25}$$

⟨*SM physics: public*⟩+≡
```
    public :: k_q_al, k_g_al
```

⟨*SM physics: procedures*⟩+≡
```
    pure function k_q_al (alpha)
      real(kind=default), intent(in) :: alpha
      real(kind=default) :: k_q_al
      k_q_al = k_q - CF * (log(alpha))**2 + gamma_q * &
                       (alpha - one - log(alpha))
    end function k_q_al

    pure function k_g_al (alpha, nf)
      real(kind=default), intent(in) :: alpha, nf
      real(kind=default) :: k_g_al
      k_g_al = k_g (nf) - CA * (log(alpha))**2 + gamma_g (nf) * &
                       (alpha - one - log(alpha))
    end function k_g_al
```

The +-distribution, but with a phase-space slicing parameter, α, $P_{1-alpha}(x) = \left(\frac{1}{1-x}\right)_{1-x}$. Since we need the fatal error message here, this function cannot be elemental.

⟨*SM physics: public*⟩+≡
```
    public :: plus_distr_al
```

⟨*SM physics: procedures*⟩+≡
```
    function plus_distr_al (x,alpha,eps) result (plusd_al)
       real(kind=default), intent(in) :: x,  eps, alpha
       real(kind=default) :: plusd_al
       if ((1.0_default - alpha) .ge. (1.0_default - eps)) then
          call msg_fatal ('sm_physics, plus_distr_al: alpha and epsilon chosen wrongly')
       elseif (x < (1.0_default - alpha)) then
          plusd_al = 0
       else if (x > (1.0_default - eps)) then
          plusd_al = log(eps/alpha)/eps
       else
          plusd_al = one/(one-x)
       end if
```

```
    end function plus_distr_al
```

Introducing phase-space slicing parameters, these flavor kernels become: The standard flavor kernels $\overline{K}^{ab}$ are:

$$\overline{K}^{qg}_\alpha(x) = \overline{K}^{\bar{q}g}_\alpha(x) = P^{qg}(x)\log(\alpha(1-x)/x) + CF \times x \qquad (4.26)$$

$$\overline{K}^{gq}_\alpha(x) = \overline{K}^{g\bar{q}}_\alpha(x) = P^{gq}(x)\log(\alpha(1-x)/x) + TR \times 2x(1-x) \qquad (4.27)$$

$$\overline{K}^{qq}_\alpha = C_F(1-x) + P^{qq}_{\text{reg}}(x)\log\frac{\alpha(1-x)}{x} + C_F\delta(1-x)\log^2\alpha + C_F\left(\frac{2}{1-x}\log\frac{1-x}{x}\right)_+ - \left(\gamma_q -\right. \qquad (4.28)$$

⟨*SM physics: public*⟩+≡
```
    public :: kbarqg_al
```

⟨*SM physics: procedures*⟩+≡
```
    function kbarqg_al (x,alpha,eps) result (kbarqgx)
      real(kind=default), intent(in) :: x, alpha, eps
      real(kind=default) :: kbarqgx
      kbarqgx = pqg (x) * log(alpha*(one-x)/x) + CF * x
    end function kbarqg_al
```

⟨*SM physics: public*⟩+≡
```
    public :: kbargq_al
```

⟨*SM physics: procedures*⟩+≡
```
    function kbargq_al (x,alpha,eps) result (kbargqx)
      real(kind=default), intent(in) :: x, alpha, eps
      real(kind=default) :: kbargqx
      kbargqx = pgq (x) * log(alpha*(one-x)/x) + two * TR * x * (one-x)
    end function kbargq_al
```

⟨*SM physics: public*⟩+≡
```
    public :: kbarqq_al
```

⟨*SM physics: procedures*⟩+≡
```
    function kbarqq_al (x,alpha,eps) result (kbarqqx)
       real(kind=default), intent(in) :: x, alpha, eps
       real(kind=default) :: kbarqqx
       kbarqqx = CF * (one - x) + pqq_reg(x) * log(alpha*(one-x)/x) &
                + CF * log_plus_distr(x,eps) &
                - (gamma_q + k_q_al(alpha) - CF * &
                five/6.0_default  * pi**2 - CF * (log(alpha))**2) * &
                delta(x,eps) + &
                CF * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
       if (x < (one-alpha)) then
          kbarqqx = kbarqqx - CF * two/(one-x) * log((two-x)/(one-x))
       end if
    end function kbarqq_al
```

⟨*SM physics: public*⟩+≡
```
    public :: kbargg_al
```

⟨*SM physics: procedures*⟩+≡
```
function kbargg_al (x,alpha,eps,nf) result (kbarggx)
   real(kind=default), intent(in) :: x, alpha, eps, nf
   real(kind=default) :: kbarggx
   kbarggx = pgg_reg(x) * log(alpha*(one-x)/x) &
             + CA * log_plus_distr(x,eps) &
             - (gamma_g(nf) + k_g_al(alpha,nf) - CA * &
             five/6.0_default  * pi**2 - CA * (log(alpha))**2) * &
             delta(x,eps) + &
             CA * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
   if (x < (one-alpha)) then
      kbarggx = kbarggx - CA * two/(one-x) * log((two-x)/(one-x))
   end if
end function kbargg_al
```

The $\tilde{K}$ flavor kernels in the presence of a phase-space slicing parameter, are:

$$\tilde{K}^{ab}(x,\alpha) = P^{qq,\mathrm{reg}}(x)\log\frac{1-x}{\alpha} + \ldots\ldots\ldots \tag{4.29}$$

⟨*SM physics: public*⟩+≡
```
public :: ktildeqq_al
```

⟨*SM physics: procedures*⟩+≡
```
function ktildeqq_al (x,alpha,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps, alpha
  real(kind=default) :: ktildeqqx
  ktildeqqx = pqq_reg(x) * log((one-x)/alpha) + CF*( &
        - log2_plus_distr_al(x,alpha,eps) - Pi**2/three * delta(x,eps) &
        + (one+x**2)/(one-x) * log(min(one,(alpha/(one-x)))) &
        + two/(one-x) * log((one+alpha-x)/alpha))
  if (x > (one-alpha)) then
     ktildeqqx = ktildeqqx - CF*two/(one-x)*log(two-x)
  end if
end function ktildeqq_al
```

This is a logarithmic +-distribution, $\left(\frac{\log((1-x)/x)}{1-x}\right)_+$. For the sampling, we need the integral over this function over the incomplete sampling interval $[0, 1-\epsilon]$, which is $\log^2(x) + 2Li_2(x) - \frac{\pi^2}{3}$. As this function is negative definite for $\epsilon > 0.1816$, we take a hard upper limit for that sampling parameter, irrespective of the fact what the user chooses.

⟨*SM physics: public*⟩+≡
```
public :: log_plus_distr
```

⟨*SM physics: procedures*⟩+≡
```
function log_plus_distr (x,eps) result (lpd)
   real(kind=default), intent(in) :: x, eps
   real(kind=default) :: lpd, eps2
   eps2 = min (eps, 0.1816_default)
   if (x > (1.0_default - eps2)) then
      lpd = ((log(eps2))**2 + two*Li2(eps2) - pi**2/three)/eps2
   else
      lpd = two*log((one-x)/x)/(one-x)
      end if
```

```
    end function log_plus_distr
```

Logarithmic +-distribution, $2 \left(\frac{\log(1/(1-x))}{1-x} \right)_+$.

⟨*SM physics: public*⟩+≡
```
    public :: log2_plus_distr
```

⟨*SM physics: procedures*⟩+≡
```
    function log2_plus_distr (x,eps) result (lpd)
        real(kind=default), intent(in) :: x, eps
        real(kind=default) :: lpd
        if (x > (1.0_default - eps)) then
           lpd = - (log(eps))**2/eps
        else
           lpd = two*log(one/(one-x))/(one-x)
        end if
    end function log2_plus_distr
```

Logarithmic +-distribution with phase-space slicing parameter, $2 \left(\frac{\log(1/(1-x))}{1-x} \right)_{1-\alpha}$.

⟨*SM physics: public*⟩+≡
```
    public :: log2_plus_distr_al
```

⟨*SM physics: procedures*⟩+≡
```
    function log2_plus_distr_al (x,alpha,eps) result (lpd_al)
      real(kind=default), intent(in) :: x, eps, alpha
      real(kind=default) :: lpd_al
      if ((1.0_default - alpha) .ge. (1.0_default - eps)) then
         call msg_fatal ('alpha and epsilon chosen wrongly')
      elseif (x < (one - alpha)) then
         lpd_al = 0
      elseif (x > (1.0_default - eps)) then
         lpd_al = - ((log(eps))**2 - (log(alpha))**2)/eps
      else
         lpd_al = two*log(one/(one-x))/(one-x)
      end if
    end function log2_plus_distr_al
```

4.4   QCD Coupling

We provide various distinct implementations of the QCD coupling. In this module, we define an abstract data type and three implementations: fixed, running with $\alpha_s(M_Z)$ as input, and running with Λ_{QCD} as input. We use the functions defined above in the module sm_physics but provide a common interface. Later modules may define additional implementations.

⟨sm_qcd.f90⟩≡
  ⟨*File header*⟩

```
  module sm_qcd
```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩

```
      use limits, only: FMT_12, MZ_REF, ALPHA_QCD_MZ_REF, LAMBDA_QCD_REF !NODEP!
      use diagnostics !NODEP!
      use md5
      use unit_tests

      use sm_physics !NODEP!
```

⟨*Standard module head*⟩

⟨*SM qcd: public*⟩

⟨*SM qcd: types*⟩

⟨*SM qcd: interfaces*⟩

```
      contains
```

⟨*SM qcd: procedures*⟩

⟨*SM qcd: tests*⟩

```
      end module sm_qcd
```

4.4.1 Coupling: Abstract Data Type

This is the abstract version of the QCD coupling implementation.

⟨*SM qcd: public*⟩≡
```
      public :: alpha_qcd_t
```

⟨*SM qcd: types*⟩≡
```
      type, abstract :: alpha_qcd_t
       contains
       ⟨SM qcd: alpha qcd: TBP⟩
      end type alpha_qcd_t
```

There must be an output routine.

⟨*SM qcd: alpha qcd: TBP*⟩≡
```
      procedure (alpha_qcd_write), deferred :: write
```

⟨*SM qcd: interfaces*⟩≡
```
      abstract interface
         subroutine alpha_qcd_write (object, unit)
           import
           class(alpha_qcd_t), intent(in) :: object
           integer, intent(in), optional :: unit
         end subroutine alpha_qcd_write
      end interface
```

This method computes the running coupling, given a certain scale. All parameters (reference value, order of the approximation, etc.) must be set before calling this.

⟨*SM qcd: alpha qcd: TBP*⟩+≡
```
      procedure (alpha_qcd_get), deferred :: get
```

⟨*SM qcd: interfaces*⟩+≡
```
abstract interface
   function alpha_qcd_get (alpha_qcd, scale) result (alpha)
     import
     class(alpha_qcd_t), intent(in) :: alpha_qcd
     real(default), intent(in) :: scale
     real(default) :: alpha
   end function alpha_qcd_get
end interface
```

4.4.2   Fixed Coupling

In this version, the α_s value is fixed, the `scale` argument of the `get` method is ignored. There is only one parameter, the value. By default, this is the value at M_Z.

⟨*SM qcd: public*⟩+≡
```
public :: alpha_qcd_fixed_t
```

⟨*SM qcd: types*⟩+≡
```
type, extends (alpha_qcd_t) :: alpha_qcd_fixed_t
   real(default) :: val = ALPHA_QCD_MZ_REF
 contains
 ⟨SM qcd: alpha qcd fixed: TBP⟩
end type alpha_qcd_fixed_t
```

Output.

⟨*SM qcd: alpha qcd fixed: TBP*⟩≡
```
procedure :: write => alpha_qcd_fixed_write
```

⟨*SM qcd: procedures*⟩≡
```
subroutine alpha_qcd_fixed_write (object, unit)
  class(alpha_qcd_fixed_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A)")  "QCD parameters (fixed coupling):"
  write (u, "(5x,A," // FMT_12 // ")")  "alpha = ", object%val
end subroutine alpha_qcd_fixed_write
```

Calculation: the scale is ignored in this case.

⟨*SM qcd: alpha qcd fixed: TBP*⟩+≡
```
procedure :: get => alpha_qcd_fixed_get
```

⟨*SM qcd: procedures*⟩+≡
```
function alpha_qcd_fixed_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_fixed_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  alpha = alpha_qcd%val
end function alpha_qcd_fixed_get
```

4.4.3 Running Coupling

In this version, the α_s value runs relative to the value at a given reference scale. There are two parameters: the value of this scale (default: M_Z), the value of α_s at this scale, and the number of effective flavors. Furthermore, we have the order of the approximation.

⟨*SM qcd: public*⟩+≡
```
public :: alpha_qcd_from_scale_t
```

⟨*SM qcd: types*⟩+≡
```
type, extends (alpha_qcd_t) :: alpha_qcd_from_scale_t
   real(default) :: mu_ref = MZ_REF
   real(default) :: ref = ALPHA_QCD_MZ_REF
   integer :: order = 0
   integer :: nf = 5
 contains
 ⟨SM qcd: alpha qcd from scale: TBP⟩
 end type alpha_qcd_from_scale_t
```

Output.

⟨*SM qcd: alpha qcd from scale: TBP*⟩≡
```
procedure :: write => alpha_qcd_from_scale_write
```

⟨*SM qcd: procedures*⟩+≡
```
subroutine alpha_qcd_from_scale_write (object, unit)
  class(alpha_qcd_from_scale_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A)")  "QCD parameters (running coupling):"
  write (u, "(5x,A," // FMT_12 // ")")  "Scale mu  = ", object%mu_ref
  write (u, "(5x,A," // FMT_12 // ")")  "alpha(mu) = ", object%ref
  write (u, "(5x,A,I0)")     "LL order  = ", object%order
  write (u, "(5x,A,I0)")     "N(flv)    = ", object%nf
end subroutine alpha_qcd_from_scale_write
```

Calculation: here, we call the function for running α_s that was defined in `sm_physics` above. The function does not take into account thresholds, so the number of flavors should be the correct one for the chosen scale. Normally, this should be the Z boson mass.

⟨*SM qcd: alpha qcd from scale: TBP*⟩+≡
```
procedure :: get => alpha_qcd_from_scale_get
```

⟨*SM qcd: procedures*⟩+≡
```
function alpha_qcd_from_scale_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_from_scale_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  alpha = running_as (scale, &
       alpha_qcd%ref, alpha_qcd%mu_ref, alpha_qcd%order, &
       real (alpha_qcd%nf, kind=default))
end function alpha_qcd_from_scale_get
```

4.4.4 Running Coupling, determined by $\Lambda_{\mathbf{QCD}}$

In this version, the input are the value Λ_{QCD} and the order of the approximation.

⟨*SM qcd: public*⟩+≡
```
public :: alpha_qcd_from_lambda_t
```

⟨*SM qcd: types*⟩+≡
```
type, extends (alpha_qcd_t) :: alpha_qcd_from_lambda_t
   real(default) :: lambda = LAMBDA_QCD_REF
   integer :: order = 0
   integer :: nf = 5
 contains
 ⟨SM qcd: alpha qcd from lambda: TBP⟩
 end type alpha_qcd_from_lambda_t
```

Output.

⟨*SM qcd: alpha qcd from lambda: TBP*⟩≡
```
procedure :: write => alpha_qcd_from_lambda_write
```

⟨*SM qcd: procedures*⟩+≡
```
subroutine alpha_qcd_from_lambda_write (object, unit)
  class(alpha_qcd_from_lambda_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A)")  "QCD parameters (Lambda_QCD as input):"
  write (u, "(5x,A," // FMT_12 // ")")  "Lambda_QCD = ", object%lambda
  write (u, "(5x,A,I0)")     "LL order   = ", object%order
  write (u, "(5x,A,I0)")     "N(flv)     = ", object%nf
end subroutine alpha_qcd_from_lambda_write
```

Calculation: here, we call the second function for running α_s that was defined in `sm_physics` above. The Λ value should be the one that is appropriate for the chosen number of effective flavors. Again, thresholds are not incorporated.

⟨*SM qcd: alpha qcd from lambda: TBP*⟩+≡
```
procedure :: get => alpha_qcd_from_lambda_get
```

⟨*SM qcd: procedures*⟩+≡
```
function alpha_qcd_from_lambda_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_from_lambda_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  alpha = running_as_lam (real (alpha_qcd%nf, kind=default), scale, &
       alpha_qcd%lambda, alpha_qcd%order)
end function alpha_qcd_from_lambda_get
```

4.4.5 Wrapper type

We could get along with a polymorphic QCD type, but a monomorphic wrapper type with a polymorphic component is easier to handle and probably safer (w.r.t. compiler bugs). However, we keep the object transparent, so we can set the type-specific parameters directly (by a `dispatch` routine).

⟨*SM qcd: public*⟩+≡

```
    public :: qcd_t
```
⟨*SM qcd: types*⟩+≡
```
    type :: qcd_t
      class(alpha_qcd_t), allocatable :: alpha
      character(32) :: md5sum = ""
     contains
     ⟨SM qcd: qcd: TBP⟩
    end type qcd_t
```

Output. We first print the polymorphic `alpha` which contains a headline, then any extra components.

⟨*SM qcd: qcd: TBP*⟩≡
```
    procedure :: write => qcd_write
```
⟨*SM qcd: procedures*⟩+≡
```
    subroutine qcd_write (qcd, unit, show_md5sum)
      class(qcd_t), intent(in) :: qcd
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: show_md5sum
      logical :: show_md5
      integer :: u
      u = output_unit (unit)
      show_md5 = .true.;  if (present (show_md5sum))  show_md5 = show_md5sum
      if (allocated (qcd%alpha)) then
         call qcd%alpha%write (u)
      else
         write (u, "(3x,A)")  "QCD parameters (coupling undefined)"
      end if
      if (show_md5 .and. qcd%md5sum /= "") &
           write (u, "(5x,A,A,A)") "md5sum = '", qcd%md5sum, "'"
    end subroutine qcd_write
```

Compute a MD5 sum for the `alpha_s` setup. This is done by writing them to a temporary file, using a standard format.

⟨*SM qcd: qcd: TBP*⟩+≡
```
    procedure :: compute_alphas_md5sum => qcd_compute_alphas_md5sum
```
⟨*SM qcd: procedures*⟩+≡
```
    subroutine qcd_compute_alphas_md5sum (qcd)
      class(qcd_t), intent(inout) :: qcd
      integer :: unit
      if (allocated (qcd%alpha)) then
         unit = free_unit ()
         open (unit, status="scratch", action="readwrite")
         call qcd%alpha%write (unit)
         rewind (unit)
         qcd%md5sum = md5sum (unit)
         close (unit)
      end if
    end subroutine qcd_compute_alphas_md5sum
```

Retrieve the MD5 sum of the qcd setup.

⟨SM qcd: qcd: TBP⟩+≡
```
procedure :: get_md5sum => qcd_get_md5sum
```

⟨SM qcd: procedures⟩+≡
```
function qcd_get_md5sum (qcd) result (md5sum)
  character(32) :: md5sum
  class(qcd_t), intent(inout) :: qcd
  md5sum = qcd%md5sum
end function qcd_get_md5sum
```

4.4.6 Unit tests

⟨SM qcd: public⟩+≡
```
public :: sm_qcd_test
```

⟨SM qcd: tests⟩≡
```
subroutine sm_qcd_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨SM qcd: execute tests⟩
```
end subroutine sm_qcd_test
```

QCD Coupling

We check two different implementations of the abstract QCD coupling.

⟨SM qcd: execute tests⟩≡
```
call test (sm_qcd_1, "sm_qcd_1", &
     "running alpha_s", &
     u, results)
```

⟨SM qcd: tests⟩+≡
```
subroutine sm_qcd_1 (u)
  integer, intent(in) :: u
  type(qcd_t) :: qcd

  write (u, "(A)")  "* Test output: sm_qcd_1"
  write (u, "(A)")  "*   Purpose: compute running alpha_s"
  write (u, "(A)")

  write (u, "(A)")  "* Fixed:"
  write (u, "(A)")

  allocate (alpha_qcd_fixed_t :: qcd%alpha)
  call qcd%compute_alphas_md5sum ()

  call qcd%write (u)
  write (u, *)
  write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
       qcd%alpha%get (MZ_REF)
  write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
       qcd%alpha%get (1000._default)
```

```
write (u, *)
deallocate (qcd%alpha)

write (u, "(A)")  "* Running from MZ (LO):"
write (u, "(A)")

allocate (alpha_qcd_from_scale_t :: qcd%alpha)
call qcd%compute_alphas_md5sum ()

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "* Running from MZ (NLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_scale_t)
   alpha%order = 1
end select
call qcd%compute_alphas_md5sum ()

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "* Running from MZ (NNLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_scale_t)
   alpha%order = 2
end select
call qcd%compute_alphas_md5sum ()

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
    qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
    qcd%alpha%get (1000._default)
write (u, *)

deallocate (qcd%alpha)
write (u, "(A)")  "* Running from Lambda_QCD (LO):"
write (u, "(A)")
```

```
      allocate (alpha_qcd_from_lambda_t :: qcd%alpha)
      call qcd%compute_alphas_md5sum ()

      call qcd%write (u)
      write (u, *)
      write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
           qcd%alpha%get (MZ_REF)
      write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
           qcd%alpha%get (1000._default)
      write (u, *)

      write (u, "(A)")  "* Running from Lambda_QCD (NLO):"
      write (u, "(A)")

      select type (alpha => qcd%alpha)
      type is (alpha_qcd_from_lambda_t)
         alpha%order = 1
      end select
      call qcd%compute_alphas_md5sum ()

      call qcd%write (u)
      write (u, *)
      write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
           qcd%alpha%get (MZ_REF)
      write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
           qcd%alpha%get (1000._default)
      write (u, *)

      write (u, "(A)")  "* Running from Lambda_QCD (NNLO):"
      write (u, "(A)")

      select type (alpha => qcd%alpha)
      type is (alpha_qcd_from_lambda_t)
         alpha%order = 2
      end select
      call qcd%compute_alphas_md5sum ()

      call qcd%write (u)
      write (u, *)
      write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
           qcd%alpha%get (MZ_REF)
      write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
           qcd%alpha%get (1000._default)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sm_qcd_1"

    end subroutine sm_qcd_1
```

Chapter 5

Physics Analysis

This part contains the structures and tools that are necessary for defining parameters, particle sets, analysis objects such as histograms, and expressions that deal with them.

These are the modules:

analysis Observables, histograms, and plots.

pdg_arrays Useful for particle aliases (e.g., 'quark' for u, d, s etc.)

prt_lists Particle lists/arrays used for analyzing events.

variables Store values of various kind, used by expressions and accessed by the command interface.

expressions Expressions of values of all kinds. Includes the API for recording analysis data.

5.1 Analysis tools

This module defines structures useful for data analysis. These include observables, histograms, and plots.

Observables are quantities that are calculated and summed up event by event. At the end, one can compute the average and error.

Histograms have their bins in addition to the observable properties. Histograms are usually written out in tables and displayed graphically.

In plots, each record creates its own entry in a table. This can be used for scatter plots if called event by event, or for plotting dependencies on parameters if called once per integration run.

Graphs are container for histograms and plots, which carry their own graphics options.

The type layout is still somewhat obfuscated. This would become much simpler if type extension could be used.

⟨`analysis.f90`⟩≡
  ⟨*File header*⟩

  ```
  module analysis
  ```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ```
    use limits, only: HISTOGRAM_HEAD_FORMAT, HISTOGRAM_DATA_FORMAT !NODEP!
    use limits, only: FMT_19 !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use os_interface
    use unit_tests
    use ifiles
  ```

  ⟨*Standard module head*⟩

  ⟨*Analysis: public*⟩

  ⟨*Analysis: parameters*⟩

  ⟨*Analysis: types*⟩

  ⟨*Analysis: interfaces*⟩

  ⟨*Analysis: variables*⟩

  ```
  contains
  ```

  ⟨*Analysis: procedures*⟩

  ⟨*Analysis: tests*⟩

  ```
  end module analysis
  ```

5.1.1 Output formats

These formats share a common field width (alignment).

⟨*Limits: public parameters*⟩+≡
```
character(*), parameter, public :: HISTOGRAM_HEAD_FORMAT = "1x,A15,3x"
character(*), parameter, public :: HISTOGRAM_INTG_FORMAT = "3x,I9,3x"
character(*), parameter, public :: HISTOGRAM_DATA_FORMAT = FMT_19
```

5.1.2 Graph options

These parameters are used for displaying data. They apply to a whole graph, which may contain more than one plot element.

The GAMELAN code chunks are part of both `graph_options` and `drawing_options`. The `drawing_options` copy is used in histograms and plots, also as graph elements. The `graph_options` copy is used for `graph` objects as a whole. Both copies are usually identical.

⟨*Analysis: public*⟩≡
```
public :: graph_options_t
```

⟨*Analysis: types*⟩≡
```
type :: graph_options_t
  private
  type(string_t) :: id
  type(string_t) :: title
  type(string_t) :: description
  type(string_t) :: x_label
  type(string_t) :: y_label
  integer :: width_mm = 130
  integer :: height_mm = 90
  logical :: x_log = .false.
  logical :: y_log = .false.
  real(default) :: x_min = 0
  real(default) :: x_max = 1
  real(default) :: y_min = 0
  real(default) :: y_max = 1
  logical :: x_min_set = .false.
  logical :: x_max_set = .false.
  logical :: y_min_set = .false.
  logical :: y_max_set = .false.
  type(string_t) :: gmlcode_bg
  type(string_t) :: gmlcode_fg
end type graph_options_t
```

Initialize the record, all strings are empty. The limits are undefined.

⟨*Analysis: public*⟩+≡
```
public :: graph_options_init
```

⟨*Analysis: procedures*⟩≡
```
subroutine graph_options_init (graph_options)
  type(graph_options_t), intent(out) :: graph_options
  graph_options%id = ""
  graph_options%title = ""
```

```
      graph_options%description = ""
      graph_options%x_label = ""
      graph_options%y_label = ""
      graph_options%gmlcode_bg = ""
      graph_options%gmlcode_fg = ""
    end subroutine graph_options_init
```

Set individual options.

⟨*Analysis: public*⟩+≡
```
    public :: graph_options_set
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_options_set (graph_options, id, &
        title, description, x_label, y_label, width_mm, height_mm, &
        x_log, y_log, x_min, x_max, y_min, y_max, &
        gmlcode_bg, gmlcode_fg)
      type(graph_options_t), intent(inout) :: graph_options
      type(string_t), intent(in), optional :: id
      type(string_t), intent(in), optional :: title
      type(string_t), intent(in), optional :: description
      type(string_t), intent(in), optional :: x_label, y_label
      integer, intent(in), optional :: width_mm, height_mm
      logical, intent(in), optional :: x_log, y_log
      real(default), intent(in), optional :: x_min, x_max, y_min, y_max
      type(string_t), intent(in), optional :: gmlcode_bg, gmlcode_fg
      if (present (id))  graph_options%id = id
      if (present (title))  graph_options%title = title
      if (present (description))  graph_options%description = description
      if (present (x_label))  graph_options%x_label = x_label
      if (present (y_label))  graph_options%y_label = y_label
      if (present (width_mm))   graph_options%width_mm  = width_mm
      if (present (height_mm))  graph_options%height_mm = height_mm
      if (present (x_log))  graph_options%x_log = x_log
      if (present (y_log))  graph_options%y_log = y_log
      if (present (x_min))  graph_options%x_min = x_min
      if (present (x_max))  graph_options%x_max = x_max
      if (present (y_min))  graph_options%y_min = y_min
      if (present (y_max))  graph_options%y_max = y_max
      if (present (x_min))  graph_options%x_min_set = .true.
      if (present (x_max))  graph_options%x_max_set = .true.
      if (present (y_min))  graph_options%y_min_set = .true.
      if (present (y_max))  graph_options%y_max_set = .true.
      if (present (gmlcode_bg))  graph_options%gmlcode_bg = gmlcode_bg
      if (present (gmlcode_fg))  graph_options%gmlcode_fg = gmlcode_fg
    end subroutine graph_options_set
```

Write a simple account of all options.

⟨*Analysis: public*⟩+≡
```
    public :: graph_options_write
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_options_write (gro, unit)
      type(graph_options_t), intent(in) :: gro
      integer, intent(in), optional :: unit
```

```
      integer :: u
      u = output_unit (unit)
1     format (A,1x,'"',A,'"')
2     format (A,1x,L1)
3     format (A,1x,ES19.12)
4     format (A,1x,I0)
5     format (A,1x,'[undefined]')
      write (u, 1)  "title       =", char (gro%title)
      write (u, 1)  "description =", char (gro%description)
      write (u, 1)  "x_label     =", char (gro%x_label)
      write (u, 1)  "y_label     =", char (gro%y_label)
      write (u, 2)  "x_log       =", gro%x_log
      write (u, 2)  "y_log       =", gro%y_log
      if (gro%x_min_set) then
         write (u, 3)  "x_min       =", gro%x_min
      else
         write (u, 5)  "x_min       ="
      end if
      if (gro%x_max_set) then
         write (u, 3)  "x_max       =", gro%x_max
      else
         write (u, 5)  "x_max       ="
      end if
      if (gro%y_min_set) then
         write (u, 3)  "y_min       =", gro%y_min
      else
         write (u, 5)  "y_min       ="
      end if
      if (gro%y_max_set) then
         write (u, 3)  "y_max       =", gro%y_max
      else
         write (u, 5)  "y_max       ="
      end if
      write (u, 4)  "width_mm    =", gro%width_mm
      write (u, 4)  "height_mm   =", gro%height_mm
      write (u, 1)  "gmlcode_bg  =", char (gro%gmlcode_bg)
      write (u, 1)  "gmlcode_fg  =", char (gro%gmlcode_fg)
   end subroutine graph_options_write
```

Write a LaTeX header/footer for the analysis file.

⟨*Analysis: procedures*⟩+≡

```
   subroutine graph_options_write_tex_header (gro, unit)
     type(graph_options_t), intent(in) :: gro
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit)
     if (gro%title /= "") then
        write (u, "(A)")
        write (u, "(A)")  "\section{" // char (gro%title) // "}"
     else
        write (u, "(A)")  "\section{" // char (quote_underscore (gro%id)) // "}"
     end if
     if (gro%description /= "") then
```

```
        write (u, "(A)")  char (gro%description)
        write (u, *)
        write (u, "(A)")  "\vspace*{\baselineskip}"
      end if
      write (u, "(A)")  "\vspace*{\baselineskip}"
      write (u, "(A)")  "\unitlength 1mm"
      write (u, "(A,I0,',',I0,A)")  &
           "\begin{gmlgraph*}(", &
           gro%width_mm, gro%height_mm, &
           ")[dat]"
    end subroutine graph_options_write_tex_header

    subroutine graph_options_write_tex_footer (gro, unit)
      type(graph_options_t), intent(in) :: gro
      integer, intent(in), optional :: unit
      integer :: u, width, height
      width = gro%width_mm - 10
      height = gro%height_mm - 10
      u = output_unit (unit)
      write (u, "(A)")  "  begingmleps ""Whizard-Logo.eps"";"
      write (u, "(A,I0,A,I0,A)")  &
           "    base := (", width, "*unitlength,", height, "*unitlength);"
      write (u, "(A)")  "    height := 9.6*unitlength;"
      write (u, "(A)")  "    width := 11.2*unitlength;"
      write (u, "(A)")  "  endgmleps;"
      write (u, "(A)")  "\end{gmlgraph*}"
    end subroutine graph_options_write_tex_footer
```

Return the analysis object ID.

⟨*Analysis: procedures*⟩+≡

```
    function graph_options_get_id (gro) result (id)
      type(string_t) :: id
      type(graph_options_t), intent(in) :: gro
      id = gro%id
    end function graph_options_get_id
```

Create an appropriate **setup** command (linear/log).

⟨*Analysis: procedures*⟩+≡

```
    function graph_options_get_gml_setup (gro) result (cmd)
      type(string_t) :: cmd
      type(graph_options_t), intent(in) :: gro
      type(string_t) :: x_str, y_str
      if (gro%x_log) then
         x_str = "log"
      else
         x_str = "linear"
      end if
      if (gro%y_log) then
         y_str = "log"
      else
         y_str = "linear"
      end if
      cmd = "setup (" // x_str // ", " // y_str // ");"
```

```
      end function graph_options_get_gml_setup
```

Return the labels in GAMELAN form.

⟨*Analysis: procedures*⟩+≡
```
    function graph_options_get_gml_x_label (gro) result (cmd)
      type(string_t) :: cmd
      type(graph_options_t), intent(in) :: gro
      cmd = 'label.bot (<' // '<' // gro%x_label // '>' // '>, out);'
    end function graph_options_get_gml_x_label

    function graph_options_get_gml_y_label (gro) result (cmd)
      type(string_t) :: cmd
      type(graph_options_t), intent(in) :: gro
      cmd = 'label.ulft (<' // '<' // gro%y_label // '>' // '>, out);'
    end function graph_options_get_gml_y_label
```

Create an appropriate `graphrange` statement for the given graph options. Where
the graph options are not set, use the supplied arguments, if any, otherwise set
the undefined value.

⟨*Analysis: procedures*⟩+≡
```
    function graph_options_get_gml_graphrange &
         (gro, x_min, x_max, y_min, y_max) result (cmd)
      type(string_t) :: cmd
      type(graph_options_t), intent(in) :: gro
      real(default), intent(in), optional :: x_min, x_max, y_min, y_max
      type(string_t) :: x_min_str, x_max_str, y_min_str, y_max_str
      character(*), parameter :: fmt = "(ES15.8)"
      if (gro%x_min_set) then
         x_min_str = "#" // trim (adjustl (real2string (gro%x_min, fmt)))
      else if (present (x_min)) then
         x_min_str = "#" // trim (adjustl (real2string (x_min, fmt)))
      else
         x_min_str = "??"
      end if
      if (gro%x_max_set) then
         x_max_str = "#" // trim (adjustl (real2string (gro%x_max, fmt)))
      else if (present (x_max)) then
         x_max_str = "#" // trim (adjustl (real2string (x_max, fmt)))
      else
         x_max_str = "??"
      end if
      if (gro%y_min_set) then
         y_min_str = "#" // trim (adjustl (real2string (gro%y_min, fmt)))
      else if (present (y_min)) then
         y_min_str = "#" // trim (adjustl (real2string (y_min, fmt)))
      else
         y_min_str = "??"
      end if
      if (gro%y_max_set) then
         y_max_str = "#" // trim (adjustl (real2string (gro%y_max, fmt)))
      else if (present (y_max)) then
         y_max_str = "#" // trim (adjustl (real2string (y_max, fmt)))
      else
```

```
        y_max_str = "??"
      end if
    cmd = "graphrange (" // x_min_str // ", " // y_min_str // "), " &
        // "(" // x_max_str // ", " // y_max_str // ");"
  end function graph_options_get_gml_graphrange
```

Get extra GAMELAN code to be executed before and after the usual drawing commands.

⟨*Analysis: procedures*⟩+≡
```
    function graph_options_get_gml_bg_command (gro) result (cmd)
      type(string_t) :: cmd
      type(graph_options_t), intent(in) :: gro
      cmd = gro%gmlcode_bg
    end function graph_options_get_gml_bg_command

    function graph_options_get_gml_fg_command (gro) result (cmd)
      type(string_t) :: cmd
      type(graph_options_t), intent(in) :: gro
      cmd = gro%gmlcode_fg
    end function graph_options_get_gml_fg_command
```

Append the header for generic data output in ifile format. We print only labels, not graphics parameters.

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_options_get_header (pl, header, comment)
      type(graph_options_t), intent(in) :: pl
      type(ifile_t), intent(inout) :: header
      type(string_t), intent(in), optional :: comment
      type(string_t) :: c
      if (present (comment)) then
        c = comment
      else
        c = ""
      end if
      call ifile_append (header, &
          c // "ID: " // pl%id)
      call ifile_append (header, &
          c // "title: " // pl%title)
      call ifile_append (header, &
          c // "description: " // pl%description)
      call ifile_append (header, &
          c // "x axis label: " // pl%x_label)
      call ifile_append (header, &
          c // "y axis label: " // pl%y_label)
    end subroutine graph_options_get_header
```

5.1.3 Drawing options

These options apply to an individual graph element (histogram or plot).

⟨*Analysis: public*⟩+≡
```
    public :: drawing_options_t
```

⟨*Analysis: types*⟩+≡
```
type :: drawing_options_t
  type(string_t) :: dataset
  logical :: with_hbars = .false.
  logical :: with_base = .false.
  logical :: piecewise = .false.
  logical :: fill = .false.
  logical :: draw = .false.
  logical :: err = .false.
  logical :: symbols = .false.
  type(string_t) :: fill_options
  type(string_t) :: draw_options
  type(string_t) :: err_options
  type(string_t) :: symbol
  type(string_t) :: gmlcode_bg
  type(string_t) :: gmlcode_fg
end type drawing_options_t
```

Write a simple account of all options.

⟨*Analysis: public*⟩+≡
```
public :: drawing_options_write
```

⟨*Analysis: procedures*⟩+≡
```
  subroutine drawing_options_write (dro, unit)
    type(drawing_options_t), intent(in) :: dro
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
1   format (A,1x,'"',A,'"')
2   format (A,1x,L1)
    write (u, 2)  "with_hbars  =", dro%with_hbars
    write (u, 2)  "with_base   =", dro%with_base
    write (u, 2)  "piecewise   =", dro%piecewise
    write (u, 2)  "fill        =", dro%fill
    write (u, 2)  "draw        =", dro%draw
    write (u, 2)  "err         =", dro%err
    write (u, 2)  "symbols     =", dro%symbols
    write (u, 1)  "fill_options=", char (dro%fill_options)
    write (u, 1)  "draw_options=", char (dro%draw_options)
    write (u, 1)  "err_options =", char (dro%err_options)
    write (u, 1)  "symbol      =", char (dro%symbol)
    write (u, 1)  "gmlcode_bg  =", char (dro%gmlcode_bg)
    write (u, 1)  "gmlcode_fg  =", char (dro%gmlcode_fg)
  end subroutine drawing_options_write
```

Init with empty strings and default options, appropriate for either histogram or plot.

⟨*Analysis: public*⟩+≡
```
public :: drawing_options_init_histogram
public :: drawing_options_init_plot
```

⟨*Analysis: procedures*⟩+≡
```
  subroutine drawing_options_init_histogram (dro)
    type(drawing_options_t), intent(out) :: dro
```

```
      dro%dataset = "dat"
      dro%with_hbars = .true.
      dro%with_base = .true.
      dro%piecewise = .true.
      dro%fill = .true.
      dro%draw = .true.
      dro%fill_options = "withcolor col.default"
      dro%draw_options = ""
      dro%err_options = ""
      dro%symbol = "fshape(circle scaled 1mm)()"
      dro%gmlcode_bg = ""
      dro%gmlcode_fg = ""
    end subroutine drawing_options_init_histogram

    subroutine drawing_options_init_plot (dro)
      type(drawing_options_t), intent(out) :: dro
      dro%dataset = "dat"
      dro%draw = .true.
      dro%fill_options = "withcolor col.default"
      dro%draw_options = ""
      dro%err_options = ""
      dro%symbol = "fshape(circle scaled 1mm)()"
      dro%gmlcode_bg = ""
      dro%gmlcode_fg = ""
    end subroutine drawing_options_init_plot
```

Set individual options.

⟨Analysis: public⟩+≡
```
    public :: drawing_options_set
```

⟨Analysis: procedures⟩+≡
```
    subroutine drawing_options_set (dro, dataset, &
        with_hbars, with_base, piecewise, fill, draw, err, symbols, &
        fill_options, draw_options, err_options, symbol, &
        gmlcode_bg, gmlcode_fg)
      type(drawing_options_t), intent(inout) :: dro
      type(string_t), intent(in), optional :: dataset
      logical, intent(in), optional :: with_hbars, with_base, piecewise
      logical, intent(in), optional :: fill, draw, err, symbols
      type(string_t), intent(in), optional :: fill_options, draw_options
      type(string_t), intent(in), optional :: err_options, symbol
      type(string_t), intent(in), optional :: gmlcode_bg, gmlcode_fg
      if (present (dataset))  dro%dataset = dataset
      if (present (with_hbars))  dro%with_hbars = with_hbars
      if (present (with_base))  dro%with_base = with_base
      if (present (piecewise))  dro%piecewise = piecewise
      if (present (fill))  dro%fill = fill
      if (present (draw))  dro%draw = draw
      if (present (err))  dro%err = err
      if (present (symbols))  dro%symbols = symbols
      if (present (fill_options))  dro%fill_options = fill_options
      if (present (draw_options))  dro%draw_options = draw_options
      if (present (err_options))  dro%err_options = err_options
      if (present (symbol))  dro%symbol = symbol
```

```
        if (present (gmlcode_bg))  dro%gmlcode_bg = gmlcode_bg
        if (present (gmlcode_fg))  dro%gmlcode_fg = gmlcode_fg
      end subroutine drawing_options_set
```

There are sepate commands for drawing the curve and for drawing errors. The symbols are applied to the latter. First of all, we may have to compute a baseline:

⟨*Analysis: procedures*⟩+≡
```
    function drawing_options_get_calc_command (dro) result (cmd)
      type(string_t) :: cmd
      type(drawing_options_t), intent(in) :: dro
      if (dro%with_base) then
         cmd = "calculate " // dro%dataset // ".base (" // dro%dataset // ") " & 
              // "(x, #0);"
      else
         cmd = ""
      end if
    end function drawing_options_get_calc_command
```

Return the drawing command.

⟨*Analysis: procedures*⟩+≡
```
    function drawing_options_get_draw_command (dro) result (cmd)
      type(string_t) :: cmd
      type(drawing_options_t), intent(in) :: dro
      if (dro%fill) then
         cmd = "fill"
      else if (dro%draw) then
         cmd = "draw"
      else
         cmd = ""
      end if
      if (dro%fill .or. dro%draw) then
         if (dro%piecewise)  cmd = cmd // " piecewise"
         if (dro%draw .and. dro%with_base)  cmd = cmd // " cyclic"
         cmd = cmd // " from (" // dro%dataset
         if (dro%with_base) then
            if (dro%piecewise) then
               cmd = cmd // ", " // dro%dataset // ".base/\"  ! "
            else
               cmd = cmd // " ~ " // dro%dataset // ".base\"  ! "
            end if
         end if
         cmd = cmd // ")"
         if (dro%fill) then
            cmd = cmd // " " // dro%fill_options
            if (dro%draw)  cmd = cmd // " outlined"
         end if
         if (dro%draw)  cmd = cmd // " " // dro%draw_options
         cmd = cmd // ";"
      end if
    end function drawing_options_get_draw_command
```

The error command draws error bars, if any.

⟨*Analysis: procedures*⟩+≡

```
function drawing_options_get_err_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  if (dro%err) then
     cmd = "draw piecewise " &
          // "from (" // dro%dataset // ".err)" &
          // " " // dro%err_options // ";"
   else
      cmd = ""
   end if
end function drawing_options_get_err_command
```

The symbol command draws symbols, if any.

⟨*Analysis: procedures*⟩+≡

```
function drawing_options_get_symb_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  if (dro%symbols) then
     cmd = "phantom" &
          // " from (" // dro%dataset // ")" &
          // " withsymbol (" // dro%symbol // ");"
   else
      cmd = ""
   end if
end function drawing_options_get_symb_command
```

Get extra GAMELAN code to be executed before and after the usual drawing commands.

⟨*Analysis: procedures*⟩+≡

```
function drawing_options_get_gml_bg_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  cmd = dro%gmlcode_bg
end function drawing_options_get_gml_bg_command

function drawing_options_get_gml_fg_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  cmd = dro%gmlcode_fg
end function drawing_options_get_gml_fg_command
```

5.1.4 Observables

The observable type holds the accumulated observable values and weight sums which are necessary for proper averaging.

⟨*Analysis: types*⟩+≡

```
type :: observable_t
  private
  real(default) :: sum_values = 0
```

```
      real(default) :: sum_squared_values = 0
      real(default) :: sum_weights = 0
      real(default) :: sum_squared_weights = 0
      integer :: count = 0
      type(string_t) :: obs_label
      type(string_t) :: obs_unit
      type(graph_options_t) :: graph_options
   end type observable_t
```

Initialize with defined properties

⟨*Analysis: procedures*⟩+≡

```
   subroutine observable_init (obs, obs_label, obs_unit, graph_options)
     type(observable_t), intent(out) :: obs
     type(string_t), intent(in), optional :: obs_label, obs_unit
     type(graph_options_t), intent(in), optional :: graph_options
     if (present (obs_label)) then
        obs%obs_label = obs_label
     else
        obs%obs_label = ""
     end if
     if (present (obs_unit)) then
        obs%obs_unit = obs_unit
     else
        obs%obs_unit = ""
     end if
     if (present (graph_options)) then
        obs%graph_options = graph_options
     else
        call graph_options_init (obs%graph_options)
     end if
   end subroutine observable_init
```

Reset all numeric entries.

⟨*Analysis: procedures*⟩+≡

```
   subroutine observable_clear (obs)
     type(observable_t), intent(inout) :: obs
     obs%sum_values = 0
     obs%sum_squared_values = 0
     obs%sum_weights = 0
     obs%sum_squared_weights = 0
     obs%count = 0
   end subroutine observable_clear
```

Record a a value. Always successful for observables.

⟨*Analysis: interfaces*⟩≡

```
   interface observable_record_value
      module procedure observable_record_value_unweighted
      module procedure observable_record_value_weighted
   end interface
```

⟨*Analysis: procedures*⟩+≡

```
   subroutine observable_record_value_unweighted (obs, value, success)
```

```
    type(observable_t), intent(inout) :: obs
    real(default), intent(in) :: value
    logical, intent(out), optional :: success
    obs%sum_values = obs%sum_values + value
    obs%sum_squared_values = obs%sum_squared_values + value**2
    obs%sum_weights = obs%sum_weights + 1
    obs%sum_squared_weights = obs%sum_squared_weights + 1
    obs%count = obs%count + 1
    if (present (success))  success = .true.
  end subroutine observable_record_value_unweighted

  subroutine observable_record_value_weighted (obs, value, weight, success)
    type(observable_t), intent(inout) :: obs
    real(default), intent(in) :: value, weight
    logical, intent(out), optional :: success
    obs%sum_values = obs%sum_values + value * weight
    obs%sum_squared_values = obs%sum_squared_values + value**2 * weight
    obs%sum_weights = obs%sum_weights + abs (weight)
    obs%sum_squared_weights = obs%sum_squared_weights + weight**2
    obs%count = obs%count + 1
    if (present (success))  success = .true.
  end subroutine observable_record_value_weighted
```

Here are the statistics formulas:

1. Unweighted case: Given a sample of n values x_i, the average is

$$\langle x \rangle = \frac{\sum x_i}{n} \tag{5.1}$$

and the error estimate

$$\Delta x = \sqrt{\frac{1}{n-1} \langle \sum (x_i - \langle x \rangle)^2} \tag{5.2}$$

$$= \sqrt{\frac{1}{n-1} \left(\frac{\sum x_i^2}{n} - \frac{(\sum x_i)^2}{n^2} \right)} \tag{5.3}$$

2. Weighted case: Instead of weight 1, each event comes with weight w_i.

$$\langle x \rangle = \frac{\sum x_i w_i}{\sum w_i} \tag{5.4}$$

and

$$\Delta x = \sqrt{\frac{1}{n-1} \left(\frac{\sum x_i^2 w_i}{\sum w_i} - \frac{(\sum x_i w_i)^2}{(\sum w_i)^2} \right)} \tag{5.5}$$

For $w_i = 1$, this specializes to the previous formula.

⟨*Analysis: procedures*⟩+≡
```
  function observable_get_n_entries (obs) result (n)
    integer :: n
    type(observable_t), intent(in) :: obs
    n = obs%count
```

```
      end function observable_get_n_entries

      function observable_get_average (obs) result (avg)
        real(default) :: avg
        type(observable_t), intent(in) :: obs
        if (obs%sum_weights /= 0) then
           avg = obs%sum_values / obs%sum_weights
        else
           avg = 0
        end if
      end function observable_get_average

      function observable_get_error (obs) result (err)
        real(default) :: err
        type(observable_t), intent(in) :: obs
        real(default) :: var, n
        if (obs%sum_weights /= 0) then
           select case (obs%count)
           case (0:1)
              err = 0
           case default
              n = obs%count
              var = obs%sum_squared_values / obs%sum_weights &
                    - (obs%sum_values / obs%sum_weights) ** 2
              err = sqrt (max (var, 0._default) / (n - 1))
           end select
        else
           err = 0
        end if
      end function observable_get_error
```

Write label and/or physical unit to a string.

⟨*Analysis: procedures*⟩+≡
```
      function observable_get_label (obs, wl, wu) result (string)
        type(string_t) :: string
        type(observable_t), intent(in) :: obs
        logical, intent(in) :: wl, wu
        type(string_t) :: obs_label, obs_unit
        if (wl) then
           if (obs%obs_label /= "") then
              obs_label = obs%obs_label
           else
              obs_label = "\textrm{Observable}"
           end if
        else
           obs_label = ""
        end if
        if (wu) then
           if (obs%obs_unit /= "") then
              if (wl) then
                 obs_unit = "\;[" // obs%obs_unit // "]"
              else
                 obs_unit = obs%obs_unit
```

```
            end if
         else
            obs_unit = ""
         end if
      else
         obs_unit = ""
      end if
      string = obs_label // obs_unit
   end function observable_get_label
```

5.1.5 Output

⟨*Analysis: procedures*⟩+≡
```
    subroutine observable_write (obs, unit)
      type(observable_t), intent(in) :: obs
      integer, intent(in), optional :: unit
      real(default) :: avg, err, relerr
      integer :: n
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      avg = observable_get_average (obs)
      err = observable_get_error (obs)
      if (avg /= 0) then
         relerr = err / avg
      else
         relerr = 0
      end if
      n = observable_get_n_entries (obs)
      if (obs%graph_options%title /= "") then
         write (u, "(A,1x,3A)") &
              "title       =", '"', char (obs%graph_options%title), '"'
      end if
      if (obs%graph_options%title /= "") then
         write (u, "(A,1x,3A)") &
              "description =", '"', char (obs%graph_options%description), '"'
      end if
      write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")", advance = "no") &
           "average     =", avg
      call write_unit ()
      write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")", advance = "no") &
           "error[abs]  =", err
      call write_unit ()
      write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
           "error[rel]  =", relerr
      write (u, "(A,1x,I0)") &
           "n_entries   =", n
    contains
      subroutine write_unit ()
        if (obs%obs_unit /= "") then
           write (u, "(1x,A)")  char (obs%obs_unit)
        else
           write (u, *)
        end if
```

```
      end subroutine write_unit
    end subroutine observable_write
```

LaTeX output.

⟨*Analysis: procedures*⟩+≡

```
    subroutine observable_write_driver (obs, unit, write_heading)
      type(observable_t), intent(in) :: obs
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: write_heading
      real(default) :: avg, err
      integer :: n_digits
      logical :: heading
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      heading = .true.;  if (present (write_heading))  heading = write_heading
      avg = observable_get_average (obs)
      err = observable_get_error (obs)
      if (avg /= 0 .and. err /= 0) then
         n_digits = max (2, 2 - int (log10 (abs (err / real (avg, default)))))
      else if (avg /= 0) then
         n_digits = 100
      else
         n_digits = 1
      end if
      if (heading) then
         write (u, "(A)")
         if (obs%graph_options%title /= "") then
            write (u, "(A)")  "\section{" // char (obs%graph_options%title) &
                 // "}"
         else
            write (u, "(A)")  "\section{Observable}"
         end if
         if (obs%graph_options%description /= "") then
            write (u, "(A)")  char (obs%graph_options%description)
            write (u, *)
         end if
         write (u, "(A)")  "\begin{flushleft}"
      end if
      write (u, "(A)", advance="no")  "  $\langle{" ! $ sign
      write (u, "(A)", advance="no")  char (observable_get_label (obs, wl=.true., wu=.false.))
      write (u, "(A)", advance="no")  "}\rangle = "
      write (u, "(A)", advance="no")  char (tex_format (avg, n_digits))
      write (u, "(A)", advance="no")  "\pm"
      write (u, "(A)", advance="no")  char (tex_format (err, 2))
      write (u, "(A)", advance="no")  "\;{"
      write (u, "(A)", advance="no")  char (observable_get_label (obs, wl=.false., wu=.true.))
      write (u, "(A)")  "}"
      write (u, "(A)", advance="no")  "     \quad[n_{\text{entries}} = "
      write (u, "(I0)",advance="no")  observable_get_n_entries (obs)
      write (u, "(A)")  "]$"           ! $ fool Emacs' noweb mode
      if (heading) then
         write (u, "(A)") "\end{flushleft}"
      end if
```

```
        end subroutine observable_write_driver
```

5.1.6 Histograms

Bins

⟨*Analysis: types*⟩+≡
```
    type :: bin_t
       private
       real(default) :: midpoint = 0
       real(default) :: width = 0
       real(default) :: sum_weights = 0
       real(default) :: sum_squared_weights = 0
       real(default) :: sum_excess_weights = 0
       integer :: count = 0
    end type bin_t
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine bin_init (bin, midpoint, width)
      type(bin_t), intent(out) :: bin
      real(default), intent(in) :: midpoint, width
      bin%midpoint = midpoint
      bin%width = width
    end subroutine bin_init
```

⟨*Analysis: procedures*⟩+≡
```
    elemental subroutine bin_clear (bin)
      type(bin_t), intent(inout) :: bin
      bin%sum_weights = 0
      bin%sum_squared_weights = 0
      bin%sum_excess_weights = 0
      bin%count = 0
    end subroutine bin_clear
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine bin_record_value (bin, normalize, weight, excess)
      type(bin_t), intent(inout) :: bin
      logical, intent(in) :: normalize
      real(default), intent(in) :: weight
      real(default), intent(in), optional :: excess
      real(default) :: w, e
      if (normalize) then
         if (bin%width /= 0) then
            w = weight / bin%width
            if (present (excess))  e = excess / bin%width
         else
            w = 0
            if (present (excess))  e = 0
         end if
      else
         w = weight
         if (present (excess))  e = excess
```

316

```
      end if
      bin%sum_weights = bin%sum_weights + abs (w)
      bin%sum_squared_weights = bin%sum_squared_weights + w ** 2
      if (present (excess)) &
           bin%sum_excess_weights = bin%sum_excess_weights + abs (e)
      bin%count = bin%count + 1
    end subroutine bin_record_value
```

⟨*Analysis: procedures*⟩+≡

```
    function bin_get_midpoint (bin) result (x)
      real(default) :: x
      type(bin_t), intent(in) :: bin
      x = bin%midpoint
    end function bin_get_midpoint

    function bin_get_width (bin) result (w)
      real(default) :: w
      type(bin_t), intent(in) :: bin
      w = bin%width
    end function bin_get_width

    function bin_get_n_entries (bin) result (n)
      integer :: n
      type(bin_t), intent(in) :: bin
      n = bin%count
    end function bin_get_n_entries

    function bin_get_sum (bin) result (s)
      real(default) :: s
      type(bin_t), intent(in) :: bin
      s = bin%sum_weights
    end function bin_get_sum

    function bin_get_error (bin) result (err)
      real(default) :: err
      type(bin_t), intent(in) :: bin
      err = sqrt (bin%sum_squared_weights)
    end function bin_get_error

    function bin_get_excess (bin) result (excess)
      real(default) :: excess
      type(bin_t), intent(in) :: bin
      excess = bin%sum_excess_weights
    end function bin_get_excess
```

⟨*Analysis: procedures*⟩+≡

```
    subroutine bin_write_header (unit)
      integer, intent(in), optional :: unit
      character(120) :: buffer
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (buffer, "(A,4(1x," //HISTOGRAM_HEAD_FORMAT // "),2x,A)") &
           "#", "bin midpoint", "value    ", "error     ", &
```

```
        "excess      ", "n"
   write (u, "(A)")  trim (buffer)
 end subroutine bin_write_header

 subroutine bin_write (bin, unit)
   type(bin_t), intent(in) :: bin
   integer, intent(in), optional :: unit
   integer :: u
   u = output_unit (unit);  if (u < 0)  return
   write (u, "(1x,4(1x," // HISTOGRAM_DATA_FORMAT // "),2x,I0)") &
        bin_get_midpoint (bin), &
        bin_get_sum (bin), &
        bin_get_error (bin), &
        bin_get_excess (bin), &
        bin_get_n_entries (bin)
 end subroutine bin_write
```

Histograms

⟨*Analysis: types*⟩+≡
```
 type :: histogram_t
   private
   real(default) :: lower_bound = 0
   real(default) :: upper_bound = 0
   real(default) :: width = 0
   integer :: n_bins = 0
   logical :: normalize_bins = .false.
   type(observable_t) :: obs
   type(observable_t) :: obs_within_bounds
   type(bin_t) :: underflow
   type(bin_t), dimension(:), allocatable :: bin
   type(bin_t) :: overflow
   type(graph_options_t) :: graph_options
   type(drawing_options_t) :: drawing_options
 end type histogram_t
```

Initializer/finalizer

Initialize a histogram. We may provide either the bin width or the number
of bins. A finalizer is not needed, since the histogram contains no pointer
(sub)components.

⟨*Analysis: interfaces*⟩+≡
```
 interface histogram_init
   module procedure histogram_init_n_bins
   module procedure histogram_init_bin_width
 end interface
```

⟨*Analysis: procedures*⟩+≡
```
 subroutine histogram_init_n_bins (h, id, &
      lower_bound, upper_bound, n_bins, normalize_bins, &
```

```
      obs_label, obs_unit, graph_options, drawing_options)
  type(histogram_t), intent(out) :: h
  type(string_t), intent(in) :: id
  real(default), intent(in) :: lower_bound, upper_bound
  integer, intent(in) :: n_bins
  logical, intent(in) :: normalize_bins
  type(string_t), intent(in), optional :: obs_label, obs_unit
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  real(default) :: bin_width
  integer :: i
  call observable_init (h%obs_within_bounds, obs_label, obs_unit)
  call observable_init (h%obs, obs_label, obs_unit)
  h%lower_bound = lower_bound
  h%upper_bound = upper_bound
  h%n_bins = max (n_bins, 1)
  h%width = h%upper_bound - h%lower_bound
  h%normalize_bins = normalize_bins
  bin_width = h%width / h%n_bins
  allocate (h%bin (h%n_bins))
  call bin_init (h%underflow, h%lower_bound, 0._default)
  do i = 1, h%n_bins
     call bin_init (h%bin(i), &
         h%lower_bound - bin_width/2 + i * bin_width, bin_width)
  end do
  call bin_init (h%overflow, h%upper_bound, 0._default)
  if (present (graph_options)) then
     h%graph_options = graph_options
  else
     call graph_options_init (h%graph_options)
  end if
  call graph_options_set (h%graph_options, id = id)
  if (present (drawing_options)) then
     h%drawing_options = drawing_options
  else
     call drawing_options_init_histogram (h%drawing_options)
  end if
end subroutine histogram_init_n_bins

subroutine histogram_init_bin_width (h, id, &
     lower_bound, upper_bound, bin_width, normalize_bins, &
     obs_label, obs_unit, graph_options, drawing_options)
  type(histogram_t), intent(out) :: h
  type(string_t), intent(in) :: id
  real(default), intent(in) :: lower_bound, upper_bound, bin_width
  logical, intent(in) :: normalize_bins
  type(string_t), intent(in), optional :: obs_label, obs_unit
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  integer :: n_bins
  if (bin_width /= 0) then
     n_bins = nint ((upper_bound - lower_bound) / bin_width)
  else
     n_bins = 1
```

```
      end if
      call histogram_init_n_bins (h, id, &
            lower_bound, upper_bound, n_bins, normalize_bins, &
            obs_label, obs_unit, graph_options, drawing_options)
   end subroutine histogram_init_bin_width
```

Initialize a histogram by copying another one.

Since h has no pointer (sub)components, intrinsic assignment is sufficient. Optionally, we replace the drawing options.

⟨*Analysis: procedures*⟩+≡
```
   subroutine histogram_init_histogram (h, h_in, drawing_options)
     type(histogram_t), intent(out) :: h
     type(histogram_t), intent(in) :: h_in
     type(drawing_options_t), intent(in), optional :: drawing_options
     h = h_in
     if (present (drawing_options)) then
        h%drawing_options = drawing_options
     end if
   end subroutine histogram_init_histogram
```

Fill histograms

Clear the histogram contents, but do not modify the structure.

⟨*Analysis: procedures*⟩+≡
```
   subroutine histogram_clear (h)
     type(histogram_t), intent(inout) :: h
     call observable_clear (h%obs)
     call observable_clear (h%obs_within_bounds)
     call bin_clear (h%underflow)
     if (allocated (h%bin))  call bin_clear (h%bin)
     call bin_clear (h%overflow)
   end subroutine histogram_clear
```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow. Optionally, we may provide an excess weight that could be returned by the unweighting procedure.

⟨*Analysis: procedures*⟩+≡
```
   subroutine histogram_record_value_unweighted (h, value, excess, success)
     type(histogram_t), intent(inout) :: h
     real(default), intent(in) :: value
     real(default), intent(in), optional :: excess
     logical, intent(out), optional :: success
     integer :: i_bin
     call observable_record_value (h%obs, value)
     if (h%width /= 0) then
        i_bin = floor (((value - h%lower_bound) / h%width) * h%n_bins) + 1
     else
        i_bin = 0
     end if
     if (i_bin <= 0) then
        call bin_record_value (h%underflow, .false., 1._default, excess)
```

```
         if (present (success))  success = .false.
      else if (i_bin <= h%n_bins) then
         call observable_record_value (h%obs_within_bounds, value)
         call bin_record_value &
               (h%bin(i_bin), h%normalize_bins, 1._default, excess)
         if (present (success))  success = .true.
      else
         call bin_record_value (h%overflow, .false., 1._default, excess)
         if (present (success))  success = .false.
      end if
   end subroutine histogram_record_value_unweighted
```

Weighted events: analogous, but no excess weight.

⟨*Analysis: procedures*⟩+≡
```
   subroutine histogram_record_value_weighted (h, value, weight, success)
     type(histogram_t), intent(inout) :: h
     real(default), intent(in) :: value, weight
     logical, intent(out), optional :: success
     integer :: i_bin
     call observable_record_value (h%obs, value, weight)
     if (h%width /= 0) then
        i_bin = floor ((((value - h%lower_bound) / h%width) * h%n_bins) + 1
     else
        i_bin = 0
     end if
     if (i_bin <= 0) then
        call bin_record_value (h%underflow, .false., weight)
        if (present (success))  success = .false.
     else if (i_bin <= h%n_bins) then
        call observable_record_value (h%obs_within_bounds, value, weight)
        call bin_record_value (h%bin(i_bin), h%normalize_bins, weight)
        if (present (success))  success = .true.
     else
        call bin_record_value (h%overflow, .false., weight)
        if (present (success))  success = .false.
     end if
   end subroutine histogram_record_value_weighted
```

Access contents

Inherited from the observable component (all-over average etc.)

⟨*Analysis: procedures*⟩+≡
```
   function histogram_get_n_entries (h) result (n)
     integer :: n
     type(histogram_t), intent(in) :: h
     n = observable_get_n_entries (h%obs)
   end function histogram_get_n_entries

   function histogram_get_average (h) result (avg)
     real(default) :: avg
     type(histogram_t), intent(in) :: h
     avg = observable_get_average (h%obs)
```

```
      end function histogram_get_average

      function histogram_get_error (h) result (err)
        real(default) :: err
        type(histogram_t), intent(in) :: h
        err = observable_get_error (h%obs)
      end function histogram_get_error
```

Analogous, but applied only to events within bounds.

⟨*Analysis: procedures*⟩+≡
```
      function histogram_get_n_entries_within_bounds (h) result (n)
        integer :: n
        type(histogram_t), intent(in) :: h
        n = observable_get_n_entries (h%obs_within_bounds)
      end function histogram_get_n_entries_within_bounds

      function histogram_get_average_within_bounds (h) result (avg)
        real(default) :: avg
        type(histogram_t), intent(in) :: h
        avg = observable_get_average (h%obs_within_bounds)
      end function histogram_get_average_within_bounds

      function histogram_get_error_within_bounds (h) result (err)
        real(default) :: err
        type(histogram_t), intent(in) :: h
        err = observable_get_error (h%obs_within_bounds)
      end function histogram_get_error_within_bounds
```

Get the number of bins

⟨*Analysis: procedures*⟩+≡
```
      function histogram_get_n_bins (h) result (n)
        type(histogram_t), intent(in) :: h
        integer :: n
        n = h%n_bins
      end function histogram_get_n_bins
```

Check bins. If the index is zero or above the limit, return the results for under-flow or overflow, respectively.

⟨*Analysis: procedures*⟩+≡
```
      function histogram_get_n_entries_for_bin (h, i) result (n)
        integer :: n
        type(histogram_t), intent(in) :: h
        integer, intent(in) :: i
        if (i <= 0) then
           n = bin_get_n_entries (h%underflow)
        else if (i <= h%n_bins) then
           n = bin_get_n_entries (h%bin(i))
        else
           n = bin_get_n_entries (h%overflow)
        end if
      end function histogram_get_n_entries_for_bin
```

```
function histogram_get_sum_for_bin (h, i) result (avg)
  real(default) :: avg
  type(histogram_t), intent(in) :: h
  integer, intent(in) :: i
  if (i <= 0) then
     avg = bin_get_sum (h%underflow)
  else if (i <= h%n_bins) then
     avg = bin_get_sum (h%bin(i))
  else
     avg = bin_get_sum (h%overflow)
  end if
end function histogram_get_sum_for_bin

function histogram_get_error_for_bin (h, i) result (err)
  real(default) :: err
  type(histogram_t), intent(in) :: h
  integer, intent(in) :: i
  if (i <= 0) then
     err = bin_get_error (h%underflow)
  else if (i <= h%n_bins) then
     err = bin_get_error (h%bin(i))
  else
     err = bin_get_error (h%overflow)
  end if
end function histogram_get_error_for_bin

function histogram_get_excess_for_bin (h, i) result (err)
  real(default) :: err
  type(histogram_t), intent(in) :: h
  integer, intent(in) :: i
  if (i <= 0) then
     err = bin_get_excess (h%underflow)
  else if (i <= h%n_bins) then
     err = bin_get_excess (h%bin(i))
  else
     err = bin_get_excess (h%overflow)
  end if
end function histogram_get_excess_for_bin
```

Return a pointer to the graph options.

⟨*Analysis: procedures*⟩+≡
```
function histogram_get_graph_options_ptr (h) result (ptr)
  type(graph_options_t), pointer :: ptr
  type(histogram_t), intent(in), target :: h
  ptr => h%graph_options
end function histogram_get_graph_options_ptr
```

Return a pointer to the drawing options.

⟨*Analysis: procedures*⟩+≡
```
function histogram_get_drawing_options_ptr (h) result (ptr)
  type(drawing_options_t), pointer :: ptr
  type(histogram_t), intent(in), target :: h
  ptr => h%drawing_options
```

```
      end function histogram_get_drawing_options_ptr
```

Output

⟨*Analysis: procedures*⟩+≡
```
    subroutine histogram_write (h, unit)
      type(histogram_t), intent(in) :: h
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call bin_write_header (u)
      if (allocated (h%bin)) then
         do i = 1, h%n_bins
            call bin_write (h%bin(i), u)
         end do
      end if
      write (u, *)
      write (u, "(A,1x,A)")  "#", "Underflow:"
      call bin_write (h%underflow, u)
      write (u, *)
      write (u, "(A,1x,A)")  "#", "Overflow:"
      call bin_write (h%overflow, u)
      write (u, *)
      write (u, "(A,1x,A)")  "#", "Summary: data within bounds"
      call observable_write (h%obs_within_bounds, u)
      write (u, *)
      write (u, "(A,1x,A)")  "#", "Summary: all data"
      call observable_write (h%obs, u)
      write (u, *)
    end subroutine histogram_write
```

Write the GAMELAN reader for histogram contents.

⟨*Analysis: procedures*⟩+≡
```
    subroutine histogram_write_gml_reader (h, filename, unit)
      type(histogram_t), intent(in) :: h
      type(string_t), intent(in) :: filename
      integer, intent(in), optional :: unit
      character(*), parameter :: fmt = "(ES15.8)"
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(2x,A)")  'fromfile "' // char (filename) // '":'
      write (u, "(4x,A)")  'key "# Histogram:";'
      write (u, "(4x,A)")  'dx := #' &
           // real2char (h%width / h%n_bins / 2, fmt) // ';'
      write (u, "(4x,A)")  'for i withinblock:'
      write (u, "(6x,A)")  'get x, y, y.d, y.n, y.e;'
      if (h%drawing_options%with_hbars) then
         write (u, "(6x,A)")  'plot (' // char (h%drawing_options%dataset) &
              // ') (x,y) hbar dx;'
      else
         write (u, "(6x,A)")  'plot (' // char (h%drawing_options%dataset) &
              // ') (x,y);'
```

```
         end if
         if (h%drawing_options%err) then
            write (u, "(6x,A)")  'plot (' // char (h%drawing_options%dataset) &
               // '.err) ' &
                  // '(x,y) vbar y.d;'
         end if
         !!! Future excess options for plots
         ! write (u, "(6x,A)")  'if show_excess: ' // &
         !            & 'plot(dat.e)(x, y plus y.e) hbar dx; fi'
         write (u, "(4x,A)")  'endfor'
         write (u, "(2x,A)")  'endfrom'
      end subroutine histogram_write_gml_reader
```

LaTeX and GAMELAN output.

⟨*Analysis: procedures*⟩+≡

```
      subroutine histogram_write_gml_driver (h, filename, unit)
        type(histogram_t), intent(in) :: h
        type(string_t), intent(in) :: filename
        integer, intent(in), optional :: unit
        type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
        integer :: u
        u = output_unit (unit);  if (u < 0)  return
        call graph_options_write_tex_header (h%graph_options, unit)
        write (u, "(2x,A)")  char (graph_options_get_gml_setup (h%graph_options))
        write (u, "(2x,A)")  char (graph_options_get_gml_graphrange &
             (h%graph_options, x_min=h%lower_bound, x_max=h%upper_bound))
        call histogram_write_gml_reader (h, filename, unit)
        calc_cmd = drawing_options_get_calc_command (h%drawing_options)
        if (calc_cmd /= "")  write (u, "(2x,A)")  char (calc_cmd)
        bg_cmd = drawing_options_get_gml_bg_command (h%drawing_options)
        if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
        draw_cmd = drawing_options_get_draw_command (h%drawing_options)
        if (draw_cmd /= "")  write (u, "(2x,A)")  char (draw_cmd)
        err_cmd = drawing_options_get_err_command (h%drawing_options)
        if (err_cmd /= "")  write (u, "(2x,A)")  char (err_cmd)
        symb_cmd = drawing_options_get_symb_command (h%drawing_options)
        if (symb_cmd /= "")  write (u, "(2x,A)")  char (symb_cmd)
        fg_cmd = drawing_options_get_gml_fg_command (h%drawing_options)
        if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
        write (u, "(2x,A)")  char (graph_options_get_gml_x_label (h%graph_options))
        write (u, "(2x,A)")  char (graph_options_get_gml_y_label (h%graph_options))
        call graph_options_write_tex_footer (h%graph_options, unit)
        write (u, "(A)") "\vspace*{2\baselineskip}"
        write (u, "(A)") "\begin{flushleft}"
        write (u, "(A)") "\textbf{Data within bounds:} \\"
        call observable_write_driver (h%obs_within_bounds, unit, &
                                      write_heading=.false.)
        write (u, "(A)") "\\[0.5\baselineskip]"
        write (u, "(A)") "\textbf{All data:} \\"
        call observable_write_driver (h%obs, unit, write_heading=.false.)
        write (u, "(A)") "\end{flushleft}"
      end subroutine histogram_write_gml_driver
```

Return the header for generic data output as an ifile.

⟨*Analysis: procedures*⟩+≡

```fortran
    subroutine histogram_get_header (h, header, comment)
      type(histogram_t), intent(in) :: h
      type(ifile_t), intent(inout) :: header
      type(string_t), intent(in), optional :: comment
      type(string_t) :: c
      if (present (comment)) then
         c = comment
      else
         c = ""
      end if
      call ifile_append (header, c // "WHIZARD histogram data")
      call graph_options_get_header (h%graph_options, header, comment)
      call ifile_append (header, &
          c // "range: " // real2string (h%lower_bound) &
          // " - " // real2string (h%upper_bound))
      call ifile_append (header, &
          c // "counts total: " &
          // int2char (histogram_get_n_entries_within_bounds (h)))
      call ifile_append (header, &
          c // "total average: " &
          // real2string (histogram_get_average_within_bounds (h)) // " +- " &
          // real2string (histogram_get_error_within_bounds (h)))
    end subroutine histogram_get_header
```

5.1.7 Plots

Points

⟨*Analysis: types*⟩+≡

```fortran
    type :: point_t
      private
      real(default) :: x = 0
      real(default) :: y = 0
      real(default) :: yerr = 0
      real(default) :: xerr = 0
      type(point_t), pointer :: next => null ()
    end type point_t
```

⟨*Analysis: interfaces*⟩+≡

```fortran
    interface point_init
      module procedure point_init_contents
      module procedure point_init_point
    end interface
```

⟨*Analysis: procedures*⟩+≡

```fortran
    subroutine point_init_contents (point, x, y, yerr, xerr)
      type(point_t), intent(out) :: point
      real(default), intent(in) :: x, y
      real(default), intent(in), optional :: yerr, xerr
      point%x = x
      point%y = y
```

```
      if (present (yerr))  point%yerr = yerr
      if (present (xerr))  point%xerr = xerr
    end subroutine point_init_contents

    subroutine point_init_point (point, point_in)
      type(point_t), intent(out) :: point
      type(point_t), intent(in) :: point_in
      point%x = point_in%x
      point%y = point_in%y
      point%yerr = point_in%yerr
      point%xerr = point_in%xerr
    end subroutine point_init_point
```

⟨*Analysis: procedures*⟩+≡

```
    function point_get_x (point) result (x)
      real(default) :: x
      type(point_t), intent(in) :: point
      x = point%x
    end function point_get_x

    function point_get_y (point) result (y)
      real(default) :: y
      type(point_t), intent(in) :: point
      y = point%y
    end function point_get_y

    function point_get_xerr (point) result (xerr)
      real(default) :: xerr
      type(point_t), intent(in) :: point
      xerr = point%xerr
    end function point_get_xerr

    function point_get_yerr (point) result (yerr)
      real(default) :: yerr
      type(point_t), intent(in) :: point
      yerr = point%yerr
    end function point_get_yerr
```

⟨*Analysis: procedures*⟩+≡

```
    subroutine point_write_header (unit)
      integer, intent(in) :: unit
      character(120) :: buffer
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (buffer, "(A,4(1x," // HISTOGRAM_HEAD_FORMAT // "))") &
           "#", "x         ", "y        ", "yerr     ", "xerr      "
      write (u, "(A)")  trim (buffer)
    end subroutine point_write_header

    subroutine point_write (point, unit)
      type(point_t), intent(in) :: point
      integer, intent(in), optional :: unit
      integer :: u
```

327

```
        u = output_unit (unit);  if (u < 0)  return
        write (u, "(1x,4(1x," // HISTOGRAM_DATA_FORMAT // "))") &
              point_get_x (point), &
              point_get_y (point), &
              point_get_yerr (point), &
              point_get_xerr (point)
    end subroutine point_write
```

Plots

⟨*Analysis: types*⟩+≡
```
    type :: plot_t
       private
       type(point_t), pointer :: first => null ()
       type(point_t), pointer :: last => null ()
       integer :: count = 0
       type(graph_options_t) :: graph_options
       type(drawing_options_t) :: drawing_options
    end type plot_t
```

Initializer/finalizer

Initialize a plot. We provide the lower and upper bound in the x direction.

⟨*Analysis: interfaces*⟩+≡
```
    interface plot_init
       module procedure plot_init_empty
       module procedure plot_init_plot
    end interface
```
⟨*Analysis: procedures*⟩+≡
```
    subroutine plot_init_empty (p, id, graph_options, drawing_options)
      type(plot_t), intent(out) :: p
      type(string_t), intent(in) :: id
      type(graph_options_t), intent(in), optional :: graph_options
      type(drawing_options_t), intent(in), optional :: drawing_options
      if (present (graph_options)) then
         p%graph_options = graph_options
      else
         call graph_options_init (p%graph_options)
      end if
      call graph_options_set (p%graph_options, id = id)
      if (present (drawing_options)) then
         p%drawing_options = drawing_options
      else
         call drawing_options_init_plot (p%drawing_options)
      end if
    end subroutine plot_init_empty
```

Initialize a plot by copying another one, optionally merging in a new set of
drawing options.

Since p has pointer (sub)components, we have to explicitly deep-copy the original.

⟨*Analysis: procedures*⟩+≡
```
subroutine plot_init_plot (p, p_in, drawing_options)
  type(plot_t), intent(out) :: p
  type(plot_t), intent(in) :: p_in
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(point_t), pointer :: current, new
  current => p_in%first
  do while (associated (current))
     allocate (new)
     call point_init (new, current)
     if (associated (p%last)) then
        p%last%next => new
     else
        p%first => new
     end if
     p%last => new
     current => current%next
  end do
  p%count = p_in%count
  p%graph_options = p_in%graph_options
  if (present (drawing_options)) then
     p%drawing_options = drawing_options
  else
     p%drawing_options = p_in%drawing_options
  end if
end subroutine plot_init_plot
```

Finalize the plot by deallocating the list of points.

⟨*Analysis: procedures*⟩+≡
```
subroutine plot_final (plot)
  type(plot_t), intent(inout) :: plot
  type(point_t), pointer :: current
  do while (associated (plot%first))
     current => plot%first
     plot%first => current%next
     deallocate (current)
  end do
  plot%last => null ()
end subroutine plot_final
```

Fill plots

Clear the plot contents, but do not modify the structure.

⟨*Analysis: procedures*⟩+≡
```
subroutine plot_clear (plot)
  type(plot_t), intent(inout) :: plot
  plot%count = 0
  call plot_final (plot)
end subroutine plot_clear
```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow.

⟨*Analysis: procedures*⟩+≡

```
subroutine plot_record_value (plot, x, y, yerr, xerr, success)
  type(plot_t), intent(inout) :: plot
  real(default), intent(in) :: x, y
  real(default), intent(in), optional :: yerr, xerr
  logical, intent(out), optional :: success
  type(point_t), pointer :: point
  plot%count = plot%count + 1
  allocate (point)
  call point_init (point, x, y, yerr, xerr)
  if (associated (plot%first)) then
     plot%last%next => point
  else
     plot%first => point
  end if
  plot%last => point
  if (present (success))  success = .true.
end subroutine plot_record_value
```

Access contents

The number of points.

⟨*Analysis: procedures*⟩+≡

```
function plot_get_n_entries (plot) result (n)
  integer :: n
  type(plot_t), intent(in) :: plot
  n = plot%count
end function plot_get_n_entries
```

Return a pointer to the graph options.

⟨*Analysis: procedures*⟩+≡

```
function plot_get_graph_options_ptr (p) result (ptr)
  type(graph_options_t), pointer :: ptr
  type(plot_t), intent(in), target :: p
  ptr => p%graph_options
end function plot_get_graph_options_ptr
```

Return a pointer to the drawing options.

⟨*Analysis: procedures*⟩+≡

```
function plot_get_drawing_options_ptr (p) result (ptr)
  type(drawing_options_t), pointer :: ptr
  type(plot_t), intent(in), target :: p
  ptr => p%drawing_options
end function plot_get_drawing_options_ptr
```

Output

This output format is used by the GAMELAN driver below.

⟨*Analysis: procedures*⟩+≡
```
subroutine plot_write (plot, unit)
  type(plot_t), intent(in) :: plot
  integer, intent(in), optional :: unit
  type(point_t), pointer :: point
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call point_write_header (u)
  point => plot%first
  do while (associated (point))
    call point_write (point, unit)
    point => point%next
  end do
  write (u, *)
  write (u, "(A,1x,A)")  "#", "Summary:"
  write (u, "(A,1x,I0)") &
      "n_entries =", plot_get_n_entries (plot)
  write (u, *)
end subroutine plot_write
```

Write the GAMELAN reader for plot contents.

⟨*Analysis: procedures*⟩+≡
```
subroutine plot_write_gml_reader (p, filename, unit)
  type(plot_t), intent(in) :: p
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(2x,A)")  'fromfile "' // char (filename) // '":'
  write (u, "(4x,A)")  'key "# Plot:";'
  write (u, "(4x,A)")  'for i withinblock:'
  write (u, "(6x,A)")  'get x, y, y.err, x.err;'
  write (u, "(6x,A)")  'plot (' // char (p%drawing_options%dataset) &
      // ') (x,y);'
  if (p%drawing_options%err) then
     write (u, "(6x,A)")  'plot (' // char (p%drawing_options%dataset) &
          // '.err) (x,y) vbar y.err hbar x.err;'
  end if
  write (u, "(4x,A)")  'endfor'
  write (u, "(2x,A)")  'endfrom'
end subroutine plot_write_gml_reader
```

LATEX and GAMELAN output. Analogous to histogram output.

⟨*Analysis: procedures*⟩+≡
```
subroutine plot_write_gml_driver (p, filename, unit)
  type(plot_t), intent(in) :: p
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
  integer :: u
```

```
        u = output_unit (unit);  if (u < 0)  return
        call graph_options_write_tex_header (p%graph_options, unit)
        write (u, "(2x,A)") &
             char (graph_options_get_gml_setup (p%graph_options))
        write (u, "(2x,A)") &
             char (graph_options_get_gml_graphrange (p%graph_options))
        call plot_write_gml_reader (p, filename, unit)
        calc_cmd = drawing_options_get_calc_command (p%drawing_options)
        if (calc_cmd /= "")  write (u, "(2x,A)")  char (calc_cmd)
        bg_cmd = drawing_options_get_gml_bg_command (p%drawing_options)
        if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
        draw_cmd = drawing_options_get_draw_command (p%drawing_options)
        if (draw_cmd /= "")  write (u, "(2x,A)")  char (draw_cmd)
        err_cmd = drawing_options_get_err_command (p%drawing_options)
        if (err_cmd /= "")  write (u, "(2x,A)")  char (err_cmd)
        symb_cmd = drawing_options_get_symb_command (p%drawing_options)
        if (symb_cmd /= "")  write (u, "(2x,A)")  char (symb_cmd)
        fg_cmd = drawing_options_get_gml_fg_command (p%drawing_options)
        if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
        write (u, "(2x,A)")  char (graph_options_get_gml_x_label (p%graph_options))
        write (u, "(2x,A)")  char (graph_options_get_gml_y_label (p%graph_options))
        call graph_options_write_tex_footer (p%graph_options, unit)
      end subroutine plot_write_gml_driver
```

Append header for generic data output in ifile format.

⟨*Analysis: procedures*⟩+≡
```
    subroutine plot_get_header (plot, header, comment)
      type(plot_t), intent(in) :: plot
      type(ifile_t), intent(inout) :: header
      type(string_t), intent(in), optional :: comment
      type(string_t) :: c
      if (present (comment)) then
         c = comment
      else
         c = ""
      end if
      call ifile_append (header, c // "WHIZARD plot data")
      call graph_options_get_header (plot%graph_options, header, comment)
      call ifile_append (header, &
           c // "number of points: " &
           // int2char (plot_get_n_entries (plot)))
    end subroutine plot_get_header
```

5.1.8 Graphs

A graph is a container for several graph elements. Each graph element is ei-
ther a plot or a histogram. There is an appropriate base type below (the
`analysis_object_t`), but to avoid recursion, we define a separate base type
here. Note that there is no actual recursion: a graph is an analysis object, but
a graph cannot contain graphs.

(If we could use type extension, the implementation would be much more
transparent.)

Graph elements

Graph elements cannot be filled by the `record` command directly. The contents are always copied from elementary histograms or plots.

⟨*Analysis: types*⟩+≡
```
type :: graph_element_t
  private
  integer :: type = AN_UNDEFINED
  type(histogram_t), pointer :: h => null ()
  type(plot_t), pointer :: p => null ()
end type graph_element_t
```

⟨*Analysis: procedures*⟩+≡
```
subroutine graph_element_final (el)
  type(graph_element_t), intent(inout) :: el
  select case (el%type)
  case (AN_HISTOGRAM)
     deallocate (el%h)
  case (AN_PLOT)
     call plot_final (el%p)
     deallocate (el%p)
  end select
  el%type = AN_UNDEFINED
end subroutine graph_element_final
```

Return the number of entries in the graph element:

⟨*Analysis: procedures*⟩+≡
```
function graph_element_get_n_entries (el) result (n)
  integer :: n
  type(graph_element_t), intent(in) :: el
  select case (el%type)
  case (AN_HISTOGRAM);  n = histogram_get_n_entries (el%h)
  case (AN_PLOT);       n = plot_get_n_entries (el%p)
  case default;         n = 0
  end select
end function graph_element_get_n_entries
```

Return a pointer to the graph / drawing options.

⟨*Analysis: procedures*⟩+≡
```
function graph_element_get_graph_options_ptr (el) result (ptr)
  type(graph_options_t), pointer :: ptr
  type(graph_element_t), intent(in) :: el
  select case (el%type)
  case (AN_HISTOGRAM);  ptr => histogram_get_graph_options_ptr (el%h)
  case (AN_PLOT);       ptr => plot_get_graph_options_ptr (el%p)
  case default;         ptr => null ()
  end select
end function graph_element_get_graph_options_ptr

function graph_element_get_drawing_options_ptr (el) result (ptr)
  type(drawing_options_t), pointer :: ptr
  type(graph_element_t), intent(in) :: el
```

```
      select case (el%type)
      case (AN_HISTOGRAM);  ptr => histogram_get_drawing_options_ptr (el%h)
      case (AN_PLOT);       ptr => plot_get_drawing_options_ptr (el%p)
      case default;         ptr => null ()
      end select
    end function graph_element_get_drawing_options_ptr
```

Output, simple wrapper for the plot/histogram writer.

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_element_write (el, unit)
      type(graph_element_t), intent(in) :: el
      integer, intent(in), optional :: unit
      type(graph_options_t), pointer :: gro
      type(string_t) :: id
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      gro => graph_element_get_graph_options_ptr (el)
      id = graph_options_get_id (gro)
      write (u, "(A,A)")  '#', repeat ("-", 78)
      select case (el%type)
      case (AN_HISTOGRAM)
         write (u, "(A)", advance="no")  "# Histogram: "
         write (u, "(1x,A)")  char (id)
         call histogram_write (el%h, unit)
      case (AN_PLOT)
         write (u, "(A)", advance="no")  "# Plot: "
         write (u, "(1x,A)")  char (id)
         call plot_write (el%p, unit)
      end select
    end subroutine graph_element_write
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_element_write_gml_reader (el, filename, unit)
      type(graph_element_t), intent(in) :: el
      type(string_t), intent(in) :: filename
      integer, intent(in), optional :: unit
      select case (el%type)
      case (AN_HISTOGRAM);  call histogram_write_gml_reader (el%h, filename, unit)
      case (AN_PLOT);       call plot_write_gml_reader (el%p, filename, unit)
      end select
    end subroutine graph_element_write_gml_reader
```

The graph type

The actual graph type contains its own `graph_options`, which override the
individual settings. The `drawing_options` are set in the graph elements. This
distinction motivates the separation of the two types.

⟨*Analysis: types*⟩+≡
```
    type :: graph_t
      private
      type(graph_element_t), dimension(:), allocatable :: el
```

```
      type(graph_options_t) :: graph_options
   end type graph_t
```

Initializer/finalizer

The graph is created with a definite number of elements. The elements are filled
one by one, optionally with modified drawing options.

⟨*Analysis: procedures*⟩+≡
```
   subroutine graph_init (g, id, n_elements, graph_options)
     type(graph_t), intent(out) :: g
     type(string_t), intent(in) :: id
     integer, intent(in) :: n_elements
     type(graph_options_t), intent(in), optional :: graph_options
     allocate (g%el (n_elements))
     if (present (graph_options)) then
        g%graph_options = graph_options
     else
        call graph_options_init (g%graph_options)
     end if
     call graph_options_set (g%graph_options, id = id)
   end subroutine graph_init
```

⟨*Analysis: procedures*⟩+≡
```
   subroutine graph_insert_histogram (g, i, h, drawing_options)
     type(graph_t), intent(inout), target :: g
     integer, intent(in) :: i
     type(histogram_t), intent(in) :: h
     type(drawing_options_t), intent(in), optional :: drawing_options
     type(graph_options_t), pointer :: gro
     type(drawing_options_t), pointer :: dro
     type(string_t) :: id
     g%el(i)%type = AN_HISTOGRAM
     allocate (g%el(i)%h)
     call histogram_init_histogram (g%el(i)%h, h, drawing_options)
     gro => histogram_get_graph_options_ptr (g%el(i)%h)
     dro => histogram_get_drawing_options_ptr (g%el(i)%h)
     id = graph_options_get_id (gro)
     call drawing_options_set (dro, dataset = "dat." // id)
   end subroutine graph_insert_histogram
```

⟨*Analysis: procedures*⟩+≡
```
   subroutine graph_insert_plot (g, i, p, drawing_options)
     type(graph_t), intent(inout) :: g
     integer, intent(in) :: i
     type(plot_t), intent(in) :: p
     type(drawing_options_t), intent(in), optional :: drawing_options
     type(graph_options_t), pointer :: gro
     type(drawing_options_t), pointer :: dro
     type(string_t) :: id
     g%el(i)%type = AN_PLOT
     allocate (g%el(i)%p)
     call plot_init_plot (g%el(i)%p, p, drawing_options)
```

```
      gro => plot_get_graph_options_ptr (g%el(i)%p)
      dro => plot_get_drawing_options_ptr (g%el(i)%p)
      id = graph_options_get_id (gro)
      call drawing_options_set (dro, dataset = "dat." // id)
    end subroutine graph_insert_plot
```

Finalizer.

⟨*Analysis: procedures*⟩+≡

```
    subroutine graph_final (g)
      type(graph_t), intent(inout) :: g
      integer :: i
      do i = 1, size (g%el)
        call graph_element_final (g%el(i))
      end do
      deallocate (g%el)
    end subroutine graph_final
```

Access contents

The number of elements.

⟨*Analysis: procedures*⟩+≡

```
    function graph_get_n_elements (graph) result (n)
      integer :: n
      type(graph_t), intent(in) :: graph
      n = size (graph%el)
    end function graph_get_n_elements
```

Retrieve a pointer to the drawing options of an element, so they can be modi-
fied. (The `target` attribute is not actually needed because the components are
pointers.)

⟨*Analysis: procedures*⟩+≡

```
    function graph_get_drawing_options_ptr (g, i) result (ptr)
      type(drawing_options_t), pointer :: ptr
      type(graph_t), intent(in), target :: g
      integer, intent(in) :: i
      ptr => graph_element_get_drawing_options_ptr (g%el(i))
    end function graph_get_drawing_options_ptr
```

Output

The default output format just writes histogram and plot data.

⟨*Analysis: procedures*⟩+≡

```
    subroutine graph_write (graph, unit)
      type(graph_t), intent(in) :: graph
      integer, intent(in), optional :: unit
      integer :: i
      do i = 1, size (graph%el)
        call graph_element_write (graph%el(i), unit)
      end do
```

```
        end subroutine graph_write
```

The GAMELAN driver is not a simple wrapper, but it writes the plot/histogram
contents embedded the complete graph. First, data are read in, global back-
ground commands next, then individual elements, then global foreground com-
mands.

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_write_gml_driver (g, filename, unit)
      type(graph_t), intent(in) :: g
      type(string_t), intent(in) :: filename
      type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
      integer, intent(in), optional :: unit
      type(drawing_options_t), pointer :: dro
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call graph_options_write_tex_header (g%graph_options, unit)
      write (u, "(2x,A)") &
           char (graph_options_get_gml_setup (g%graph_options))
      write (u, "(2x,A)") &
           char (graph_options_get_gml_graphrange (g%graph_options))
      do i = 1, size (g%el)
         call graph_element_write_gml_reader (g%el(i), filename, unit)
         calc_cmd = drawing_options_get_calc_command &
                       (graph_element_get_drawing_options_ptr (g%el(i)))
         if (calc_cmd /= "")  write (u, "(2x,A)")  char (calc_cmd)
      end do
      bg_cmd = graph_options_get_gml_bg_command (g%graph_options)
      if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
      do i = 1, size (g%el)
         dro => graph_element_get_drawing_options_ptr (g%el(i))
         bg_cmd = drawing_options_get_gml_bg_command (dro)
         if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
         draw_cmd = drawing_options_get_draw_command (dro)
         if (draw_cmd /= "")  write (u, "(2x,A)")  char (draw_cmd)
         err_cmd = drawing_options_get_err_command (dro)
         if (err_cmd /= "")  write (u, "(2x,A)")  char (err_cmd)
         symb_cmd = drawing_options_get_symb_command (dro)
         if (symb_cmd /= "")  write (u, "(2x,A)")  char (symb_cmd)
         fg_cmd = drawing_options_get_gml_fg_command (dro)
         if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
      end do
      fg_cmd = graph_options_get_gml_fg_command (g%graph_options)
      if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
      write (u, "(2x,A)")  char (graph_options_get_gml_x_label (g%graph_options))
      write (u, "(2x,A)")  char (graph_options_get_gml_y_label (g%graph_options))
      call graph_options_write_tex_footer (g%graph_options, unit)
    end subroutine graph_write_gml_driver
```

Append header for generic data output in ifile format.

⟨*Analysis: procedures*⟩+≡
```
    subroutine graph_get_header (graph, header, comment)
      type(graph_t), intent(in) :: graph
      type(ifile_t), intent(inout) :: header
```

```
type(string_t), intent(in), optional :: comment
type(string_t) :: c
if (present (comment)) then
   c = comment
else
   c = ""
end if
call ifile_append (header, c // "WHIZARD graph data")
call graph_options_get_header (graph%graph_options, header, comment)
call ifile_append (header, &
      c // "number of graph elements: " &
      // int2char (graph_get_n_elements (graph)))
end subroutine graph_get_header
```

5.1.9   Analysis objects

This data structure holds all observables, histograms and such that are currently active. We have one global store; individual items are identified by their ID strings.

(This should rather be coded by type extension.)

⟨*Analysis: parameters*⟩≡
```
integer, parameter :: AN_UNDEFINED = 0
integer, parameter :: AN_OBSERVABLE = 1
integer, parameter :: AN_HISTOGRAM = 2
integer, parameter :: AN_PLOT = 3
integer, parameter :: AN_GRAPH = 4
```

⟨*Analysis: public*⟩+≡
```
public :: AN_UNDEFINED, AN_HISTOGRAM, AN_OBSERVABLE, AN_PLOT, AN_GRAPH
```

⟨*Analysis: types*⟩+≡
```
type :: analysis_object_t
   private
   type(string_t) :: id
   integer :: type = AN_UNDEFINED
   type(observable_t), pointer :: obs => null ()
   type(histogram_t), pointer :: h => null ()
   type(plot_t), pointer :: p => null ()
   type(graph_t), pointer :: g => null ()
   type(analysis_object_t), pointer :: next => null ()
end type analysis_object_t
```

Initializer/finalizer

Allocate with the correct type but do not fill initial values.

⟨*Analysis: procedures*⟩+≡
```
subroutine analysis_object_init (obj, id, type)
  type(analysis_object_t), intent(out) :: obj
  type(string_t), intent(in) :: id
  integer, intent(in) :: type
  obj%id = id
```

```
      obj%type = type
      select case (obj%type)
      case (AN_OBSERVABLE);  allocate (obj%obs)
      case (AN_HISTOGRAM);   allocate (obj%h)
      case (AN_PLOT);        allocate (obj%p)
      case (AN_GRAPH);       allocate (obj%g)
      end select
    end subroutine analysis_object_init
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_final (obj)
      type(analysis_object_t), intent(inout) :: obj
      select case (obj%type)
      case (AN_OBSERVABLE)
         deallocate (obj%obs)
      case (AN_HISTOGRAM)
         deallocate (obj%h)
      case (AN_PLOT)
         call plot_final (obj%p)
         deallocate (obj%p)
      case (AN_GRAPH)
         call graph_final (obj%g)
         deallocate (obj%g)
      end select
      obj%type = AN_UNDEFINED
    end subroutine analysis_object_final
```

Clear the analysis object, i.e., reset it to its initial state. Not applicable to graphs, which are always combinations of other existing objects.

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_clear (obj)
      type(analysis_object_t), intent(inout) :: obj
      select case (obj%type)
      case (AN_OBSERVABLE)
         call observable_clear (obj%obs)
      case (AN_HISTOGRAM)
         call histogram_clear (obj%h)
      case (AN_PLOT)
         call plot_clear (obj%p)
      end select
    end subroutine analysis_object_clear
```

Fill with data

Record data. The effect depends on the type of analysis object.

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_record_data (obj, &
         x, y, yerr, xerr, weight, excess, success)
      type(analysis_object_t), intent(inout) :: obj
      real(default), intent(in) :: x
      real(default), intent(in), optional :: y, yerr, xerr, weight, excess
```

```
      logical, intent(out), optional :: success
      select case (obj%type)
      case (AN_OBSERVABLE)
         if (present (weight)) then
            call observable_record_value_weighted (obj%obs, x, weight, success)
         else
            call observable_record_value_unweighted (obj%obs, x, success)
         end if
      case (AN_HISTOGRAM)
         if (present (weight)) then
            call histogram_record_value_weighted (obj%h, x, weight, success)
         else
            call histogram_record_value_unweighted (obj%h, x, excess, success)
         end if
      case (AN_PLOT)
         if (present (y)) then
            call plot_record_value (obj%p, x, y, yerr, xerr, success)
         else
            if (present (success))  success = .false.
         end if
      case default
         if (present (success))  success = .false.
      end select
    end subroutine analysis_object_record_data
```

Explicitly set the pointer to the next object in the list.

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_object_set_next_ptr (obj, next)
      type(analysis_object_t), intent(inout) :: obj
      type(analysis_object_t), pointer :: next
      obj%next => next
    end subroutine analysis_object_set_next_ptr
```

Access contents

Return a pointer to the next object in the list.

⟨*Analysis: procedures*⟩+≡
```
    function analysis_object_get_next_ptr (obj) result (next)
      type(analysis_object_t), pointer :: next
      type(analysis_object_t), intent(in) :: obj
      next => obj%next
    end function analysis_object_get_next_ptr
```

Return data as appropriate for the object type.

⟨*Analysis: procedures*⟩+≡
```
    function analysis_object_get_n_elements (obj) result (n)
      integer :: n
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_HISTOGRAM)
         n = 1
```

```fortran
    case (AN_PLOT)
       n = 1
    case (AN_GRAPH)
       n = graph_get_n_elements (obj%g)
    case default
       n = 0
    end select
end function analysis_object_get_n_elements

function analysis_object_get_n_entries (obj, within_bounds) result (n)
  integer :: n
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
  select case (obj%type)
  case (AN_OBSERVABLE)
     n = observable_get_n_entries (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        n = histogram_get_n_entries_within_bounds (obj%h)
     else
        n = histogram_get_n_entries (obj%h)
     end if
  case (AN_PLOT)
     n = plot_get_n_entries (obj%p)
  case default
     n = 0
  end select
end function analysis_object_get_n_entries

function analysis_object_get_average (obj, within_bounds) result (avg)
  real(default) :: avg
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
  select case (obj%type)
  case (AN_OBSERVABLE)
     avg = observable_get_average (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        avg = histogram_get_average_within_bounds (obj%h)
     else
        avg = histogram_get_average (obj%h)
     end if
  case default
     avg = 0
  end select
end function analysis_object_get_average

function analysis_object_get_error (obj, within_bounds) result (err)
  real(default) :: err
  type(analysis_object_t), intent(in) :: obj
```

```
      logical, intent(in), optional :: within_bounds
      logical :: wb
      select case (obj%type)
      case (AN_OBSERVABLE)
         err = observable_get_error (obj%obs)
      case (AN_HISTOGRAM)
         wb = .false.;  if (present (within_bounds)) wb = within_bounds
         if (wb) then
            err = histogram_get_error_within_bounds (obj%h)
         else
            err = histogram_get_error (obj%h)
         end if
      case default
         err = 0
      end select
    end function analysis_object_get_error
```

Return pointers to the actual contents:

⟨*Analysis: procedures*⟩+≡

```
    function analysis_object_get_observable_ptr (obj) result (obs)
      type(observable_t), pointer :: obs
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_OBSERVABLE);  obs => obj%obs
      case default;          obs => null ()
      end select
    end function analysis_object_get_observable_ptr

    function analysis_object_get_histogram_ptr (obj) result (h)
      type(histogram_t), pointer :: h
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_HISTOGRAM);  h => obj%h
      case default;         h => null ()
      end select
    end function analysis_object_get_histogram_ptr

    function analysis_object_get_plot_ptr (obj) result (plot)
      type(plot_t), pointer :: plot
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_PLOT);  plot => obj%p
      case default;    plot => null ()
      end select
    end function analysis_object_get_plot_ptr

    function analysis_object_get_graph_ptr (obj) result (g)
      type(graph_t), pointer :: g
      type(analysis_object_t), intent(in) :: obj
      select case (obj%type)
      case (AN_GRAPH);  g => obj%g
      case default;     g => null ()
      end select
    end function analysis_object_get_graph_ptr
```

Return true if the object has a graphical representation:

⟨*Analysis: procedures*⟩+≡

```
function analysis_object_has_plot (obj) result (flag)
  logical :: flag
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_HISTOGRAM);  flag = .true.
  case (AN_PLOT);       flag = .true.
  case (AN_GRAPH);      flag = .true.
  case default;         flag = .false.
  end select
end function analysis_object_has_plot
```

Output

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_object_write (obj, unit, verbose)
  type(analysis_object_t), intent(in) :: obj
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  verb = .false.;  if (present (verbose))  verb = verbose
  write (u, "(A)")  repeat ("#", 79)
  select case (obj%type)
  case (AN_OBSERVABLE)
     write (u, "(A)", advance="no")  "# Observable:"
  case (AN_HISTOGRAM)
     write (u, "(A)", advance="no")  "# Histogram: "
  case (AN_PLOT)
     write (u, "(A)", advance="no")  "# Plot: "
  case (AN_GRAPH)
     write (u, "(A)", advance="no")  "# Graph: "
  case default
     write (u, "(A)") "# [undefined analysis object]"
     return
  end select
  write (u, "(1x,A)")  char (obj%id)
  select case (obj%type)
  case (AN_OBSERVABLE)
     call observable_write (obj%obs, unit)
  case (AN_HISTOGRAM)
     if (verb) then
        call graph_options_write (obj%h%graph_options, unit)
        write (u, *)
        call drawing_options_write (obj%h%drawing_options, unit)
        write (u, *)
     end if
     call histogram_write (obj%h, unit)
  case (AN_PLOT)
```

```
      if (verb) then
         call graph_options_write (obj%p%graph_options, unit)
         write (u, *)
         call drawing_options_write (obj%p%drawing_options, unit)
         write (u, *)
      end if
      call plot_write (obj%p, unit)
   case (AN_GRAPH)
      call graph_write (obj%g, unit)
   end select
end subroutine analysis_object_write
```

Write the object part of the LaTeX driver file.

⟨*Analysis: procedures*⟩+≡
```
subroutine analysis_object_write_driver (obj, filename, unit)
  type(analysis_object_t), intent(in) :: obj
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  select case (obj%type)
  case (AN_OBSERVABLE)
     call observable_write_driver (obj%obs, unit)
  case (AN_HISTOGRAM)
     call histogram_write_gml_driver (obj%h, filename, unit)
  case (AN_PLOT)
     call plot_write_gml_driver (obj%p, filename, unit)
  case (AN_GRAPH)
     call graph_write_gml_driver (obj%g, filename, unit)
  end select
end subroutine analysis_object_write_driver
```

Return a data header for external formats, in ifile form.

⟨*Analysis: procedures*⟩+≡
```
subroutine analysis_object_get_header (obj, header, comment)
  type(analysis_object_t), intent(in) :: obj
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  select case (obj%type)
  case (AN_HISTOGRAM)
     call histogram_get_header (obj%h, header, comment)
  case (AN_PLOT)
     call plot_get_header (obj%p, header, comment)
  end select
end subroutine analysis_object_get_header
```

5.1.10   Analysis object iterator

Analysis objects are containers which have iterable data structures: histograms/bins
and plots/points. If they are to be treated on a common basis, it is useful to
have an iterator which hides the implementation details.

   The iterator is used only for elementary analysis objects that contain plot

data: histograms or plots. It is invalid for meta-objects (graphs) and non-graphical objects (observables).

⟨*Analysis: public*⟩+≡
```
public :: analysis_iterator_t
```

⟨*Analysis: types*⟩+≡
```
type :: analysis_iterator_t
  private
  integer :: type = AN_UNDEFINED
  type(analysis_object_t), pointer :: object => null ()
  integer :: index = 1
  type(point_t), pointer :: point => null ()
end type
```

The initializer places the iterator at the beginning of the analysis object.

⟨*Analysis: procedures*⟩+≡
```
subroutine analysis_iterator_init (iterator, object)
  type(analysis_iterator_t), intent(out) :: iterator
  type(analysis_object_t), intent(in), target :: object
  iterator%object => object
  if (associated (iterator%object)) then
     iterator%type = iterator%object%type
     select case (iterator%type)
     case (AN_PLOT)
        iterator%point => iterator%object%p%first
     end select
  end if
end subroutine analysis_iterator_init
```

The iterator is valid as long as it points to an existing entry. An iterator for a data object without array data (observable) is always invalid.

⟨*Analysis: public*⟩+≡
```
public :: analysis_iterator_is_valid
```

⟨*Analysis: procedures*⟩+≡
```
function analysis_iterator_is_valid (iterator) result (valid)
  logical :: valid
  type(analysis_iterator_t), intent(in) :: iterator
  if (associated (iterator%object)) then
     select case (iterator%type)
     case (AN_HISTOGRAM)
        valid = iterator%index <= histogram_get_n_bins (iterator%object%h)
     case (AN_PLOT)
        valid = associated (iterator%point)
     case default
        valid = .false.
     end select
  else
     valid = .false.
  end if
end function analysis_iterator_is_valid
```

Advance the iterator.

⟨Analysis: public⟩+≡
```
    public :: analysis_iterator_advance
```

⟨Analysis: procedures⟩+≡
```
    subroutine analysis_iterator_advance (iterator)
      type(analysis_iterator_t), intent(inout) :: iterator
      if (associated (iterator%object)) then
         select case (iterator%type)
         case (AN_PLOT)
            iterator%point => iterator%point%next
         end select
         iterator%index = iterator%index + 1
      end if
    end subroutine analysis_iterator_advance
```

Retrieve the object type:

⟨Analysis: public⟩+≡
```
    public :: analysis_iterator_get_type
```

⟨Analysis: procedures⟩+≡
```
    function analysis_iterator_get_type (iterator) result (type)
      integer :: type
      type(analysis_iterator_t), intent(in) :: iterator
      type = iterator%type
    end function analysis_iterator_get_type
```

Use the iterator to retrieve data. We implement a common routine which takes the data descriptors as optional arguments. Data which do not occur in the selected type trigger to an error condition.

The iterator must point to a valid entry.

⟨Analysis: public⟩+≡
```
    public :: analysis_iterator_get_data
```

⟨Analysis: procedures⟩+≡
```
    subroutine analysis_iterator_get_data (iterator, &
        x, y, yerr, xerr, width, excess, index, n_total)
      type(analysis_iterator_t), intent(in) :: iterator
      real(default), intent(out), optional :: x, y, yerr, xerr, width, excess
      integer, intent(out), optional :: index, n_total
      select case (iterator%type)
      case (AN_HISTOGRAM)
         if (present (x)) &
              x = bin_get_midpoint (iterator%object%h%bin(iterator%index))
         if (present (y)) &
              y = bin_get_sum (iterator%object%h%bin(iterator%index))
         if (present (yerr)) &
              yerr = bin_get_error (iterator%object%h%bin(iterator%index))
         if (present (xerr)) &
              call invalid ("histogram", "xerr")
         if (present (width)) &
              width = bin_get_width (iterator%object%h%bin(iterator%index))
         if (present (excess)) &
              excess = bin_get_excess (iterator%object%h%bin(iterator%index))
```

```
      if (present (index)) &
           index = iterator%index
      if (present (n_total)) &
           n_total = histogram_get_n_bins (iterator%object%h)
   case (AN_PLOT)
      if (present (x)) &
           x = point_get_x (iterator%point)
      if (present (y)) &
           y = point_get_y (iterator%point)
      if (present (yerr)) &
           yerr = point_get_yerr (iterator%point)
      if (present (xerr)) &
           xerr = point_get_xerr (iterator%point)
      if (present (width)) &
           call invalid ("plot", "width")
      if (present (excess)) &
           call invalid ("plot", "excess")
      if (present (index)) &
           index = iterator%index
      if (present (n_total)) &
           n_total = plot_get_n_entries (iterator%object%p)
   case default
      call msg_bug ("analysis_iterator_get_data: called " &
           // "for unsupported analysis object type")
   end select
 contains
   subroutine invalid (typestr, objstr)
     character(*), intent(in) :: typestr, objstr
     call msg_bug ("analysis_iterator_get_data: attempt to get '" &
          // objstr // "' for type '" // typestr // "'")
   end subroutine invalid
 end subroutine analysis_iterator_get_data
```

5.1.11   Analysis store

This data structure holds all observables, histograms and such that are currently
active. We have one global store; individual items are identified by their ID
strings and types.

⟨Analysis: variables⟩≡
```
    type(analysis_store_t), save :: analysis_store
```

⟨Analysis: types⟩+≡
```
    type :: analysis_store_t
      private
      type(analysis_object_t), pointer :: first => null ()
      type(analysis_object_t), pointer :: last => null ()
    end type analysis_store_t
```

Delete the analysis store
⟨Analysis: public⟩+≡
```
    public :: analysis_final
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_final ()
      type(analysis_object_t), pointer :: current
      do while (associated (analysis_store%first))
         current => analysis_store%first
         analysis_store%first => current%next
         call analysis_object_final (current)
      end do
      analysis_store%last => null ()
    end subroutine analysis_final
```

Append a new analysis object

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_store_append_object (id, type)
      type(string_t), intent(in) :: id
      integer, intent(in) :: type
      type(analysis_object_t), pointer :: obj
      allocate (obj)
      call analysis_object_init (obj, id, type)
      if (associated (analysis_store%last)) then
         analysis_store%last%next => obj
      else
         analysis_store%first => obj
      end if
      analysis_store%last => obj
    end subroutine analysis_store_append_object
```

Return a pointer to the analysis object with given ID.

⟨*Analysis: procedures*⟩+≡
```
    function analysis_store_get_object_ptr (id) result (obj)
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      obj => analysis_store%first
      do while (associated (obj))
         if (obj%id == id)  return
         obj => obj%next
      end do
    end function analysis_store_get_object_ptr
```

Initialize an analysis object: either reset it if present, or append a new entry.

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_store_init_object (id, type, obj)
      type(string_t), intent(in) :: id
      integer, intent(in) :: type
      type(analysis_object_t), pointer :: obj, next
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         next => analysis_object_get_next_ptr (obj)
         call analysis_object_final (obj)
         call analysis_object_init (obj, id, type)
         call analysis_object_set_next_ptr (obj, next)
      else
```

```
      call analysis_store_append_object (id, type)
      obj => analysis_store%last
    end if
  end subroutine analysis_store_init_object
```

Get the type of a analysis object

⟨*Analysis: public*⟩+≡
```
    public :: analysis_store_get_object_type
```

⟨*Analysis: procedures*⟩+≡
```
    function analysis_store_get_object_type (id) result (type)
      type(string_t), intent(in) :: id
      integer :: type
      type(analysis_object_t), pointer :: object
      object => analysis_store_get_object_ptr (id)
      if (associated (object)) then
        type = object%type
      else
        type = AN_UNDEFINED
      end if
    end function analysis_store_get_object_type
```

Return the number of objects in the store.

⟨*Analysis: procedures*⟩+≡
```
    function analysis_store_get_n_objects () result (n)
      integer :: n
      type(analysis_object_t), pointer :: current
      n = 0
      current => analysis_store%first
      do while (associated (current))
        n = n + 1
        current => current%next
      end do
    end function analysis_store_get_n_objects
```

Allocate an array and fill it with all existing IDs.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_store_get_ids
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_store_get_ids (id)
      type(string_t), dimension(:), allocatable, intent(out) :: id
      type(analysis_object_t), pointer :: current
      integer :: i
      allocate (id (analysis_store_get_n_objects()))
      i = 0
      current => analysis_store%first
      do while (associated (current))
        i = i + 1
        id(i) = current%id
        current => current%next
      end do
    end subroutine analysis_store_get_ids
```

5.1.12   LᴬTEX driver file

Write a driver file for all objects in the store.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_write_driver_all (filename_data, unit)
  type(string_t), intent(in) :: filename_data
  integer, intent(in), optional :: unit
  type(analysis_object_t), pointer :: obj
  call analysis_store_write_driver_header (unit)
  obj => analysis_store%first
  do while (associated (obj))
     call analysis_object_write_driver (obj, filename_data, unit)
     obj => obj%next
  end do
  call analysis_store_write_driver_footer (unit)
end subroutine analysis_store_write_driver_all
```

Write a driver file for an array of objects.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_write_driver_obj (filename_data, id, unit)
  type(string_t), intent(in) :: filename_data
  type(string_t), dimension(:), intent(in) :: id
  integer, intent(in), optional :: unit
  type(analysis_object_t), pointer :: obj
  integer :: i
  call analysis_store_write_driver_header (unit)
  do i = 1, size (id)
     obj => analysis_store_get_object_ptr (id(i))
     if (associated (obj))  &
          call analysis_object_write_driver (obj, filename_data, unit)
  end do
  call analysis_store_write_driver_footer (unit)
end subroutine analysis_store_write_driver_obj
```

The beginning of the driver file.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_write_driver_header (unit)
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, '(A)') "\documentclass[12pt]{article}"
  write (u, *)
  write (u, '(A)') "\usepackage{gamelan}"
  write (u, '(A)') "\usepackage{amsmath}"
  write (u, '(A)') "\usepackage{ifpdf}"
  write (u, '(A)') "\ifpdf"
  write (u, '(A)') "   \DeclareGraphicsRule{*}{mps}{*}{}"
  write (u, '(A)') "\else"
  write (u, '(A)') "   \DeclareGraphicsRule{*}{eps}{*}{}"
  write (u, '(A)') "\fi"
  write (u, *)
  write (u, '(A)') "\begin{document}"
  write (u, '(A)') "\begin{gmlfile}"
```

```
    write (u, *)
    write (u, '(A)') "\begin{gmlcode}"
    write (u, '(A)') "  color col.default, col.excess;"
    write (u, '(A)') "  col.default = 0.9white;"
    write (u, '(A)') "  col.excess  = red;"
    write (u, '(A)') "  boolean show_excess;"
    !!! Future excess options for plots
    ! if (mcs(1)%plot_excess .and. mcs(1)%unweighted) then
    !    write (u, '(A)') "  show_excess = true;"
    ! else
    write (u, '(A)') "  show_excess = false;"
    ! end if
    write (u, '(A)') "\end{gmlcode}"
    write (u, *)
  end subroutine analysis_store_write_driver_header
```

The end of the driver file.

⟨*Analysis: procedures*⟩+≡
```
  subroutine analysis_store_write_driver_footer (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit);  if (u < 0)  return
    write(u, *)
    write(u, '(A)') "\end{gmlfile}"
    write(u, '(A)') "\end{document}"
  end subroutine analysis_store_write_driver_footer
```

5.1.13   API

Creating new objects

The specific versions below:

⟨*Analysis: public*⟩+≡
```
  public :: analysis_init_observable
```

⟨*Analysis: procedures*⟩+≡
```
  subroutine analysis_init_observable (id, obs_label, obs_unit, graph_options)
    type(string_t), intent(in) :: id
    type(string_t), intent(in), optional :: obs_label, obs_unit
    type(graph_options_t), intent(in), optional :: graph_options
    type(analysis_object_t), pointer :: obj
    type(observable_t), pointer :: obs
    call analysis_store_init_object (id, AN_OBSERVABLE, obj)
    obs => analysis_object_get_observable_ptr (obj)
    call observable_init (obs, obs_label, obs_unit, graph_options)
  end subroutine analysis_init_observable
```

⟨*Analysis: public*⟩+≡
```
  public :: analysis_init_histogram
```

⟨*Analysis: interfaces*⟩+≡
```
  interface analysis_init_histogram
     module procedure analysis_init_histogram_n_bins
```

```
      module procedure analysis_init_histogram_bin_width
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_init_histogram_n_bins &
        (id, lower_bound, upper_bound, n_bins, normalize_bins, &
         obs_label, obs_unit, graph_options, drawing_options)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: lower_bound, upper_bound
      integer, intent(in) :: n_bins
      logical, intent(in) :: normalize_bins
      type(string_t), intent(in), optional :: obs_label, obs_unit
      type(graph_options_t), intent(in), optional :: graph_options
      type(drawing_options_t), intent(in), optional :: drawing_options
      type(analysis_object_t), pointer :: obj
      type(histogram_t), pointer :: h
      call analysis_store_init_object (id, AN_HISTOGRAM, obj)
      h => analysis_object_get_histogram_ptr (obj)
      call histogram_init (h, id, &
          lower_bound, upper_bound, n_bins, normalize_bins, &
          obs_label, obs_unit, graph_options, drawing_options)
    end subroutine analysis_init_histogram_n_bins

    subroutine analysis_init_histogram_bin_width &
        (id, lower_bound, upper_bound, bin_width, normalize_bins, &
         obs_label, obs_unit, graph_options, drawing_options)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: lower_bound, upper_bound, bin_width
      logical, intent(in) :: normalize_bins
      type(string_t), intent(in), optional :: obs_label, obs_unit
      type(graph_options_t), intent(in), optional :: graph_options
      type(drawing_options_t), intent(in), optional :: drawing_options
      type(analysis_object_t), pointer :: obj
      type(histogram_t), pointer :: h
      call analysis_store_init_object (id, AN_HISTOGRAM, obj)
      h => analysis_object_get_histogram_ptr (obj)
      call histogram_init (h, id, &
          lower_bound, upper_bound, bin_width, normalize_bins, &
          obs_label, obs_unit, graph_options, drawing_options)
    end subroutine analysis_init_histogram_bin_width
```

⟨*Analysis: public*⟩+≡
```
    public :: analysis_init_plot
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_init_plot (id, graph_options, drawing_options)
      type(string_t), intent(in) :: id
      type(graph_options_t), intent(in), optional :: graph_options
      type(drawing_options_t), intent(in), optional :: drawing_options
      type(analysis_object_t), pointer :: obj
      type(plot_t), pointer :: plot
      call analysis_store_init_object (id, AN_PLOT, obj)
      plot => analysis_object_get_plot_ptr (obj)
      call plot_init (plot, id, graph_options, drawing_options)
```

```
      end subroutine analysis_init_plot
```

⟨*Analysis: public*⟩+≡
```
    public :: analysis_init_graph
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_init_graph (id, n_elements, graph_options)
      type(string_t), intent(in) :: id
      integer, intent(in) :: n_elements
      type(graph_options_t), intent(in), optional :: graph_options
      type(analysis_object_t), pointer :: obj
      type(graph_t), pointer :: graph
      call analysis_store_init_object (id, AN_GRAPH, obj)
      graph => analysis_object_get_graph_ptr (obj)
      call graph_init (graph, id, n_elements, graph_options)
    end subroutine analysis_init_graph
```

Recording data

This procedure resets an object or the whole store to its initial state.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_clear
```

⟨*Analysis: interfaces*⟩+≡
```
    interface analysis_clear
       module procedure analysis_store_clear_obj
       module procedure analysis_store_clear_all
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_store_clear_obj (id)
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         call analysis_object_clear (obj)
      end if
    end subroutine analysis_store_clear_obj

    subroutine analysis_store_clear_all ()
      type(analysis_object_t), pointer :: obj
      obj => analysis_store%first
      do while (associated (obj))
         call analysis_object_clear (obj)
         obj => obj%next
      end do
    end subroutine analysis_store_clear_all
```

There is one generic recording function whose behavior depends on the type of analysis object.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_record_data
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_record_data (id, x, y, yerr, xerr, &
        weight, excess, success, exist)
      type(string_t), intent(in) :: id
      real(default), intent(in) :: x
      real(default), intent(in), optional :: y, yerr, xerr, weight, excess
      logical, intent(out), optional :: success, exist
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         call analysis_object_record_data (obj, x, y, yerr, xerr, &
              weight, excess, success)
         if (present (exist))  exist = .true.
      else
         if (present (success))  success = .false.
         if (present (exist))  exist = .false.
      end if
    end subroutine analysis_record_data
```

Build a graph

This routine sets up the array of graph elements by copying the graph elements
given as input. The object must exist and already be initialized as a graph.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_fill_graph
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_fill_graph (id, i, id_in, drawing_options)
      type(string_t), intent(in) :: id
      integer, intent(in) :: i
      type(string_t), intent(in) :: id_in
      type(drawing_options_t), intent(in), optional :: drawing_options
      type(analysis_object_t), pointer :: obj
      type(graph_t), pointer :: g
      type(histogram_t), pointer :: h
      type(plot_t), pointer :: p
      obj => analysis_store_get_object_ptr (id)
      g => analysis_object_get_graph_ptr (obj)
      obj => analysis_store_get_object_ptr (id_in)
      if (associated (obj)) then
         select case (obj%type)
         case (AN_HISTOGRAM)
            h => analysis_object_get_histogram_ptr (obj)
            call graph_insert_histogram (g, i, h, drawing_options)
         case (AN_PLOT)
            p => analysis_object_get_plot_ptr (obj)
            call graph_insert_plot (g, i, p, drawing_options)
         case default
            call msg_error ("Graph '" // char (id) // "': Element '" &
                 // char (id_in) // "' is neither histogram nor plot.")
         end select
      else
         call msg_error ("Graph '" // char (id) // "': Element '" &
```

354

```
                // char (id_in) // "' is undefined.")
        end if
    end subroutine analysis_fill_graph
```

Retrieve generic results

Check if a named object exists.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_exists
```

⟨*Analysis: procedures*⟩+≡
```
    function analysis_exists (id) result (flag)
      type(string_t), intent(in) :: id
      logical :: flag
      type(analysis_object_t), pointer :: obj
      flag = .true.
      obj => analysis_store%first
      do while (associated (obj))
         if (obj%id == id)  return
         obj => obj%next
      end do
      flag = .false.
    end function analysis_exists
```

The following functions should work for all kinds of analysis object:

⟨*Analysis: public*⟩+≡
```
    public :: analysis_get_n_elements
    public :: analysis_get_n_entries
    public :: analysis_get_average
    public :: analysis_get_error
```

⟨*Analysis: procedures*⟩+≡
```
    function analysis_get_n_elements (id) result (n)
      integer :: n
      type(string_t), intent(in) :: id
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         n = analysis_object_get_n_elements (obj)
      else
         n = 0
      end if
    end function analysis_get_n_elements

    function analysis_get_n_entries (id, within_bounds) result (n)
      integer :: n
      type(string_t), intent(in) :: id
      logical, intent(in), optional :: within_bounds
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         n = analysis_object_get_n_entries (obj, within_bounds)
      else
```

```
      n = 0
    end if
  end function analysis_get_n_entries

  function analysis_get_average (id, within_bounds) result (avg)
    real(default) :: avg
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    logical, intent(in), optional :: within_bounds
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
       avg = analysis_object_get_average (obj, within_bounds)
    else
       avg = 0
    end if
  end function analysis_get_average

  function analysis_get_error (id, within_bounds) result (err)
    real(default) :: err
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    logical, intent(in), optional :: within_bounds
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
       err = analysis_object_get_error (obj, within_bounds)
    else
       err = 0
    end if
  end function analysis_get_error
```

Return true if any analysis object is graphical
⟨*Analysis: public*⟩+≡
```
    public :: analysis_has_plots
```

⟨*Analysis: interfaces*⟩+≡
```
    interface analysis_has_plots
       module procedure analysis_has_plots_any
       module procedure analysis_has_plots_obj
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    function analysis_has_plots_any () result (flag)
      logical :: flag
      type(analysis_object_t), pointer :: obj
      flag = .false.
      obj => analysis_store%first
      do while (associated (obj))
         flag = analysis_object_has_plot (obj)
         if (flag)  return
      end do
    end function analysis_has_plots_any

    function analysis_has_plots_obj (id) result (flag)
      logical :: flag
```

```
    type(string_t), dimension(:), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    integer :: i
    flag = .false.
    do i = 1, size (id)
       obj => analysis_store_get_object_ptr (id(i))
       if (associated (obj)) then
          flag = analysis_object_has_plot (obj)
          if (flag)  return
       end if
    end do
  end function analysis_has_plots_obj
```

Iterators

Initialize an iterator for the given object. If the object does not exist or has wrong type, the iterator will be invalid.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_init_iterator
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_init_iterator (id, iterator)
      type(string_t), intent(in) :: id
      type(analysis_iterator_t), intent(out) :: iterator
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj))  call analysis_iterator_init (iterator, obj)
    end subroutine analysis_init_iterator
```

Output

⟨*Analysis: public*⟩+≡
```
    public :: analysis_write
```

⟨*Analysis: interfaces*⟩+≡
```
    interface analysis_write
       module procedure analysis_write_object
       module procedure analysis_write_all
    end interface
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_write_object (id, unit, verbose)
      type(string_t), intent(in) :: id
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      type(analysis_object_t), pointer :: obj
      obj => analysis_store_get_object_ptr (id)
      if (associated (obj)) then
         call analysis_object_write (obj, unit, verbose)
      else
         call msg_error ("Analysis object '" // char (id) // "' not found")
      end if
```

```
      end subroutine analysis_write_object

   subroutine analysis_write_all (unit, verbose)
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose
     type(analysis_object_t), pointer :: obj
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     obj => analysis_store%first
     do while (associated (obj))
        call analysis_object_write (obj, unit, verbose)
        obj => obj%next
     end do
   end subroutine analysis_write_all
```

⟨*Analysis: public*⟩+≡
```
   public :: analysis_write_driver
```
⟨*Analysis: procedures*⟩+≡
```
   subroutine analysis_write_driver (filename_data, id, unit)
     type(string_t), intent(in) :: filename_data
     type(string_t), dimension(:), intent(in), optional :: id
     integer, intent(in), optional :: unit
     if (present (id)) then
        call analysis_store_write_driver_obj (filename_data, id, unit)
     else
        call analysis_store_write_driver_all (filename_data, unit)
     end if
   end subroutine analysis_write_driver
```

⟨*Analysis: public*⟩+≡
```
   public :: analysis_compile_tex
```
⟨*Analysis: procedures*⟩+≡
```
   subroutine analysis_compile_tex (file, has_gmlcode, os_data)
     type(string_t), intent(in) :: file
     logical, intent(in) :: has_gmlcode
     type(os_data_t), intent(in) :: os_data
     type(string_t) :: setenv
     integer :: status
     if (os_data%event_analysis_ps) then
        BLOCK: do
           if (os_data%whizard_texpath /= "") then
              setenv = "TEXINPUTS=" // os_data%whizard_texpath // ":$TEXINPUTS "
           else
              setenv = ""
           end if
           call os_system_call (setenv // os_data%latex // " " // file, status)
           if (status /= 0)  exit BLOCK
           if (has_gmlcode) then
              call os_system_call (os_data%gml // " " // file, status)
              if (status /= 0)  exit BLOCK
              call os_system_call (setenv // os_data%latex // " " // file, &
                   status)
              if (status /= 0)  exit BLOCK
```

```
             end if
             call os_system_call (os_data%dvips // " -o " // file // ".ps " &
                  // file, status)
             if (status /= 0)  exit BLOCK
             if (os_data%event_analysis_pdf) then
                call os_system_call (os_data%ps2pdf // " " // file // ".ps", &
                                     status)
                if (status /= 0)  exit BLOCK
             end if
             exit BLOCK
          end do BLOCK
          if (status /= 0) then
             call msg_error ("Unable to compile analysis output file")
          end if
       else
          call msg_warning ("Skipping results display because " &
                // "latex/mpost/dvips is not available")
       end if
    end subroutine analysis_compile_tex
```

Write header for generic data output to an ifile.

⟨*Analysis: public*⟩+≡
```
    public :: analysis_get_header
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_get_header (id, header, comment)
      type(string_t), intent(in) :: id
      type(ifile_t), intent(inout) :: header
      type(string_t), intent(in), optional :: comment
      type(analysis_object_t), pointer :: object
      object => analysis_store_get_object_ptr (id)
      if (associated (object)) then
         call analysis_object_get_header (object, header, comment)
      end if
    end subroutine analysis_get_header
```

5.1.14 Unit tests

⟨*Analysis: public*⟩+≡
```
    public :: analysis_test
```

⟨*Analysis: procedures*⟩+≡
```
    subroutine analysis_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Analysis: execute tests*⟩
```
    end subroutine analysis_test
```

⟨*Analysis: execute tests*⟩≡
```
    call test (analysis_1, "analysis_1", &
        "check elementary analysis building blocks", &
        u, results)
```

⟨Analysis: tests⟩≡

```
subroutine analysis_1 (u)
  integer, intent(in) :: u
  type(string_t) :: id1, id2, id3, id4
  integer :: i
  id1 = "foo"
  id2 = "bar"
  id3 = "hist"
  id4 = "plot"

  write (u, "(A)")  "* Test output: Analysis"
  write (u, "(A)")  "*   Purpose: test the analysis routines"
  write (u, "(A)")

  call analysis_init_observable (id1)
  call analysis_init_observable (id2)
  call analysis_init_histogram_bin_width &
       (id3, 0.5_default, 5.5_default, 1._default, normalize_bins=.false.)
  call analysis_init_plot (id4)
  do i = 1, 3
     write (u, "(A,1x," // FMT_19 // ")")  "data = ", real(i,default)
     call analysis_record_data (id1, real(i,default))
     call analysis_record_data (id2, real(i,default), &
                                    weight=real(i,default))
     call analysis_record_data (id3, real(i,default))
     call analysis_record_data (id4, real(i,default), real(i,default)**2)
  end do
  write (u, "(A,10(1x,I5))") "n_entries = ", &
       analysis_get_n_entries (id1), &
       analysis_get_n_entries (id2), &
       analysis_get_n_entries (id3), &
       analysis_get_n_entries (id3, within_bounds = .true.), &
       analysis_get_n_entries (id4), &
       analysis_get_n_entries (id4, within_bounds = .true.)
  write (u, "(A,10(1x," // FMT_19 // "))")  "average  = ", &
       analysis_get_average (id1), &
       analysis_get_average (id2), &
       analysis_get_average (id3), &
       analysis_get_average (id3, within_bounds = .true.)
  write (u, "(A,10(1x," // FMT_19 // "))")  "error    = ", &
       analysis_get_error (id1), &
       analysis_get_error (id2), &
       analysis_get_error (id3), &
       analysis_get_error (id3, within_bounds = .true.)

  write (u, "(A)")
  write (u, "(A)") "* Clear analysis #2"
  write (u, "(A)")

  call analysis_clear (id2)
  do i = 4, 6
     print *, "data = ", real(i,default)
     call analysis_record_data (id1, real(i,default))
     call analysis_record_data (id2, real(i,default), &
```

```
                               weight=real(i,default))
     call analysis_record_data (id3, real(i,default))
     call analysis_record_data (id4, real(i,default), real(i,default)**2)
  end do
  write (u, "(A,10(1x,I5))")  "n_entries = ", &
       analysis_get_n_entries (id1), &
       analysis_get_n_entries (id2), &
       analysis_get_n_entries (id3), &
       analysis_get_n_entries (id3, within_bounds = .true.), &
       analysis_get_n_entries (id4), &
       analysis_get_n_entries (id4, within_bounds = .true.)
  write (u, "(A,10(1x," // FMT_19 // "))")  "average  = ", &
       analysis_get_average (id1), &
       analysis_get_average (id2), &
       analysis_get_average (id3), &
       analysis_get_average (id3, within_bounds = .true.)
  write (u, "(A,10(1x," // FMT_19 // "))")  "error    = ", &
       analysis_get_error (id1), &
       analysis_get_error (id2), &
       analysis_get_error (id3), &
       analysis_get_error (id3, within_bounds = .true.)
  write (u, "(A)")
  call analysis_write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call analysis_clear ()
  call analysis_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: analysis_1"
end subroutine analysis_1
```

5.2 Fastjet interface

The FastJet library is linked externally, if available. The wrapper code is also in a separate directory. Here, we define WHIZARD-specific procedures and tests.

⟨jets.f90⟩≡

  ⟨*File header*⟩

  ```
  module jets
  ```

  ⟨*Use kinds*⟩
  ```
    use unit_tests
    use fastjet !NODEP!
  ```

  ⟨*Standard module head*⟩

  ⟨*Jets: public*⟩

  ```
  contains
  ```

  ⟨*Jets: procedures*⟩

  ⟨*Jets: tests*⟩

  ```
  end module jets
  ```

5.2.1 Re-exported symbols

We use this module as a proxy for the FastJet interface, therefore we re-export some symbols.

⟨*Jets: public*⟩≡
```
    public :: fastjet_available
    public :: fastjet_init
    public :: jet_definition_t
    public :: pseudojet_t
    public :: pseudojet_vector_t
    public :: cluster_sequence_t
    public :: assignment (=)
```

The initialization routine prints the banner.

⟨*Jets: procedures*⟩≡
```
    subroutine fastjet_init ()
      call print_banner ()
    end subroutine fastjet_init
```

The jet algorithm codes (actually, integers)

⟨*Jets: public*⟩+≡
```
    public :: kt_algorithm
    public :: cambridge_algorithm
    public :: antikt_algorithm
    public :: genkt_algorithm
    public :: cambridge_for_passive_algorithm
    public :: genkt_for_passive_algorithm
```

```
    public :: ee_kt_algorithm
    public :: ee_genkt_algorithm
    public :: plugin_algorithm
    public :: undefined_jet_algorithm
```

5.2.2 Unit tests

⟨*Jets: public*⟩+≡
```
    public :: jets_test
```

⟨*Jets: procedures*⟩+≡
```
    subroutine jets_test (u, results)
      integer, intent(in) :: u
      type (test_results_t), intent(inout) :: results
```
⟨*Jets: execute tests*⟩
```
    end subroutine jets_test
```

This test is actually the minimal example from the FastJet manual, translated to Fortran.

Note that FastJet creates pseudojet vectors, which we mirror in the `pseudojet_vector_t`, but immediately assign to pseudojet arrays. Without automatic finalization available in the compilers, we should avoid this in actual code and rather introduce intermediate variables for those objects, which we can finalize explicitly.

⟨*Jets: execute tests*⟩≡
```
    call test (jets_1, "jets_1", &
         "basic FastJet functionality", &
         u, results)
```

⟨*Jets: tests*⟩≡
```
    subroutine jets_1 (u)
      integer, intent(in) :: u

      type(pseudojet_t), dimension(:), allocatable :: prt, jets, constituents
      type(jet_definition_t) :: jet_def
      type(cluster_sequence_t) :: cs

      integer, parameter :: dp = default
      integer :: i, j

      write (u, "(A)")  "* Test output: jets_1"
      write (u, "(A)")  "*   Purpose: test basic FastJet functionality"
      write (u, "(A)")

      write (u, "(A)")  "* Print banner"
      call print_banner ()

      write (u, *)
      write (u, "(A)")  "* Prepare input particles"
      allocate (prt (3))
      call prt(1)%init ( 99._dp, 0.1_dp, 0._dp, 100._dp)
      call prt(2)%init (  4._dp,-0.1_dp, 0._dp,   5._dp)
      call prt(3)%init (-99._dp, 0._dp,  0._dp,  99._dp)
```

```
   write (u, *)
   write (u, "(A)")  "* Define jet algorithm"
   call jet_def%init (antikt_algorithm, 0.7_dp)

   write (u, *)
   write (u, "(A)")  "* Cluster particles according to jet algorithm"

   write (u, *)
   write (u, "(A,A)")  "Clustering with ", jet_def%description ()
   call cs%init (pseudojet_vector (prt), jet_def)

   write (u, *)
   write (u, "(A)")  "* Sort output jets"
   jets = sorted_by_pt (cs%inclusive_jets ())

   write (u, *)
   write (u, "(A)")  "* Print jet observables and constituents"
   write (u, *)
   write (u, "(4x,3(7x,A3))") "pt", "y", "phi"
   do i = 1, size (jets)
      write (u, "(A,1x,I0,A,3(1x,F9.5))") &
           "jet", i, ":", jets(i)%perp (), jets(i)%rap (), jets(i)%phi ()
      constituents = jets(i)%constituents ()
      do j = 1, size (constituents)
         write (u, "(4x,A,1x,I0,A,F9.5)") &
              "constituent", j, "'s pt:", constituents(j)%perp ()
      end do
      do j = 1, size (constituents)
         call constituents(j)%final ()
      end do
   end do

   write (u, *)
   write (u, "(A)")  "* Cleanup"

   do i = 1, size (prt)
      call prt(i)%final ()
   end do
   do i = 1, size (jets)
      call jets(i)%final ()
   end do
   call jet_def%final ()
   call cs%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: jets_1"

end subroutine jets_1
```

5.3   PDG arrays

For defining aliases, we introduce a special type which holds a set of (integer) PDG codes.

⟨pdg_arrays.f90⟩≡
  ⟨*File header*⟩

  module pdg_arrays

  ⟨*Use file utils*⟩
    use sorting
    use unit_tests

  ⟨*Standard module head*⟩

  ⟨*PDG arrays: public*⟩

  ⟨*PDG arrays: parameters*⟩

  ⟨*PDG arrays: types*⟩

  ⟨*PDG arrays: interfaces*⟩

  contains

  ⟨*PDG arrays: procedures*⟩

  ⟨*PDG arrays: tests*⟩

  end module pdg_arrays

5.3.1   Type definition

Using an allocatable array eliminates the need for initializer and/or finalizer.

⟨*PDG arrays: public*⟩≡
    public :: pdg_array_t

⟨*PDG arrays: types*⟩≡
    type :: pdg_array_t
       private
       integer, dimension(:), allocatable :: pdg
     contains
     ⟨*PDG arrays: pdg array: TBP*⟩
    end type pdg_array_t

Output

⟨*PDG arrays: public*⟩+≡
    public :: pdg_array_write

⟨*PDG arrays: pdg array: TBP*⟩≡
    procedure :: write => pdg_array_write

⟨PDG arrays: procedures⟩≡

```
subroutine pdg_array_write (aval, unit)
  class(pdg_array_t), intent(in) :: aval
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(A)", advance="no")  "PDG("
  if (allocated (aval%pdg)) then
     do i = 1, size (aval%pdg)
        if (i > 1)  write (u, "(A)", advance="no")  ", "
        write (u, "(I0)", advance="no")  aval%pdg(i)
     end do
  end if
  write (u, "(A)", advance="no")  ")"
end subroutine pdg_array_write
```

5.3.2   Parameters

We need an UNDEFINED value:

⟨PDG arrays: parameters⟩≡

```
integer, parameter, public :: UNDEFINED = 0
```

5.3.3   Basic operations

Assignment. We define assignment from and to an integer array. Note that the integer array, if it is the l.h.s., must be declared allocatable by the caller.

⟨PDG arrays: public⟩+≡

```
public :: assignment(=)
```

⟨PDG arrays: interfaces⟩≡

```
interface assignment(=)
   module procedure pdg_array_from_int_array
   module procedure pdg_array_from_int
   module procedure int_array_from_pdg_array
end interface
```

⟨PDG arrays: procedures⟩+≡

```
subroutine pdg_array_from_int_array (aval, iarray)
  type(pdg_array_t), intent(out) :: aval
  integer, dimension(:), intent(in) :: iarray
  allocate (aval%pdg (size (iarray)))
  aval%pdg = iarray
end subroutine pdg_array_from_int_array

elemental subroutine pdg_array_from_int (aval, int)
  type(pdg_array_t), intent(out) :: aval
  integer, intent(in) :: int
  allocate (aval%pdg (1))
  aval%pdg = int
end subroutine pdg_array_from_int

subroutine int_array_from_pdg_array (iarray, aval)
```

```
      integer, dimension(:), allocatable, intent(out) :: iarray
      type(pdg_array_t), intent(in) :: aval
      if (allocated (aval%pdg)) then
         allocate (iarray (size (aval%pdg)))
         iarray = aval%pdg
      else
         allocate (iarray (0))
      end if
   end subroutine int_array_from_pdg_array
```

Length of the array.

⟨*PDG arrays: public*⟩+≡
```
   public :: pdg_array_get_length
```

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
   procedure :: get_length => pdg_array_get_length
```

⟨*PDG arrays: procedures*⟩+≡
```
   elemental function pdg_array_get_length (aval) result (n)
     class(pdg_array_t), intent(in) :: aval
     integer :: n
     if (allocated (aval%pdg)) then
        n = size (aval%pdg)
     else
        n = 0
     end if
   end function pdg_array_get_length
```

Return the element with index i.

⟨*PDG arrays: public*⟩+≡
```
   public :: pdg_array_get
```

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
   procedure :: get => pdg_array_get
```

⟨*PDG arrays: procedures*⟩+≡
```
   elemental function pdg_array_get (aval, i) result (pdg)
     class(pdg_array_t), intent(in) :: aval
     integer, intent(in) :: i
     integer :: pdg
     pdg = aval%pdg(i)
   end function pdg_array_get
```

Replace element with index i by a new array of elements.

⟨*PDG arrays: public*⟩+≡
```
   public :: pdg_array_replace
```

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
   procedure :: replace => pdg_array_replace
```

⟨*PDG arrays: procedures*⟩+≡
```
   function pdg_array_replace (aval, i, pdg_new) result (aval_new)
     class(pdg_array_t), intent(in) :: aval
     integer, intent(in) :: i
     integer, dimension(:), intent(in) :: pdg_new
```

```
      type(pdg_array_t) :: aval_new
      integer :: n, l
      n = size (aval%pdg)
      l = size (pdg_new)
      allocate (aval_new%pdg (n + l - 1))
      aval_new%pdg(:i-1) = aval%pdg(:i-1)
      aval_new%pdg(i:i+l-1) = pdg_new
      aval_new%pdg(i+l:) = aval%pdg(i+1:)
    end function pdg_array_replace
```

Concatenate two PDG arrays

⟨*PDG arrays: public*⟩+≡
```
    public :: operator(//)
```

⟨*PDG arrays: interfaces*⟩+≡
```
    interface operator(//)
       module procedure concat_pdg_arrays
    end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
    function concat_pdg_arrays (aval1, aval2) result (aval)
      type(pdg_array_t) :: aval
      type(pdg_array_t), intent(in) :: aval1, aval2
      integer :: n1, n2
      if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
         n1 = size (aval1%pdg)
         n2 = size (aval2%pdg)
         allocate (aval%pdg (n1 + n2))
         aval%pdg(:n1) = aval1%pdg
         aval%pdg(n1+1:) = aval2%pdg
      else if (allocated (aval1%pdg)) then
         aval = aval1
      else if (allocated (aval2%pdg)) then
         aval = aval2
      end if
    end function concat_pdg_arrays
```

5.3.4 Matching

A PDG array matches a given PDG code if the code is present within the array.
If either one is zero (UNDEFINED), the match also succeeds.

⟨*PDG arrays: public*⟩+≡
```
    public :: operator(.match.)
```

⟨*PDG arrays: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure pdg_array_match_integer
    end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
    elemental function pdg_array_match_integer (aval, pdg) result (flag)
      logical :: flag
      type(pdg_array_t), intent(in) :: aval
      integer, intent(in) :: pdg
      if (allocated (aval%pdg)) then
         flag = pdg == UNDEFINED &
               .or. any (aval%pdg == UNDEFINED) &
               .or. any (aval%pdg == pdg)
      else
         flag = .false.
      end if
    end function pdg_array_match_integer
```

Comparison. Here, we take the PDG arrays as-is, assuming that they are sorted.

The ordering is a bit odd: first, we look only at the absolute values of the PDG codes. If they all match, the particle comes before the antiparticle, scanning from left to right.

⟨*PDG arrays: public*⟩+≡
```
    public :: operator(<)
    public :: operator(>)
    public :: operator(<=)
    public :: operator(>=)
    public :: operator(==)
    public :: operator(/=)
```

⟨*PDG arrays: interfaces*⟩+≡
```
    interface operator(<)
       module procedure pdg_array_lt
    end interface
    interface operator(>)
       module procedure pdg_array_gt
    end interface
    interface operator(<=)
       module procedure pdg_array_le
    end interface
    interface operator(>=)
       module procedure pdg_array_ge
    end interface
    interface operator(==)
       module procedure pdg_array_eq
    end interface
    interface operator(/=)
       module procedure pdg_array_ne
    end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
    elemental function pdg_array_lt (aval1, aval2) result (flag)
      type(pdg_array_t), intent(in) :: aval1, aval2
      logical :: flag
      integer :: i
      if (size (aval1%pdg) /= size (aval2%pdg)) then
         flag = size (aval1%pdg) < size (aval2%pdg)
```

```
        else
            do i = 1, size (aval1%pdg)
                if (abs (aval1%pdg(i)) /= abs (aval2%pdg(i))) then
                    flag = abs (aval1%pdg(i)) < abs (aval2%pdg(i))
                    return
                end if
            end do
            do i = 1, size (aval1%pdg)
                if (aval1%pdg(i) /= aval2%pdg(i)) then
                    flag = aval1%pdg(i) > aval2%pdg(i)
                    return
                end if
            end do
            flag = .false.
        end if
    end function pdg_array_lt

    elemental function pdg_array_gt (aval1, aval2) result (flag)
        type(pdg_array_t), intent(in) :: aval1, aval2
        logical :: flag
        flag = .not. (aval1 < aval2 .or. aval1 == aval2)
    end function pdg_array_gt

    elemental function pdg_array_le (aval1, aval2) result (flag)
        type(pdg_array_t), intent(in) :: aval1, aval2
        logical :: flag
        flag = aval1 < aval2 .or. aval1 == aval2
    end function pdg_array_le

    elemental function pdg_array_ge (aval1, aval2) result (flag)
        type(pdg_array_t), intent(in) :: aval1, aval2
        logical :: flag
        flag = .not. (aval1 < aval2)
    end function pdg_array_ge

    elemental function pdg_array_eq (aval1, aval2) result (flag)
        type(pdg_array_t), intent(in) :: aval1, aval2
        logical :: flag
        if (size (aval1%pdg) /= size (aval2%pdg)) then
            flag = .false.
        else
            flag = all (aval1%pdg == aval2%pdg)
        end if
    end function pdg_array_eq

    elemental function pdg_array_ne (aval1, aval2) result (flag)
        type(pdg_array_t), intent(in) :: aval1, aval2
        logical :: flag
        flag = .not. (aval1 == aval2)
    end function pdg_array_ne
```

Equivalence. Two PDG arrays are equivalent if either one contains UNDEFINED or if each element of array 1 is present in array 2, and vice versa.

⟨*PDG arrays: public*⟩+≡
```
    public :: operator(.eqv.)
    public :: operator(.neqv.)
```
⟨*PDG arrays: interfaces*⟩+≡
```
    interface operator(.eqv.)
       module procedure pdg_array_equivalent
    end interface
    interface operator(.neqv.)
       module procedure pdg_array_inequivalent
    end interface
```

⟨*PDG arrays: procedures*⟩+≡
```
    elemental function pdg_array_equivalent (aval1, aval2) result (eq)
      logical :: eq
      type(pdg_array_t), intent(in) :: aval1, aval2
      logical, dimension(:), allocatable :: match1, match2
      integer :: i
      if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
         eq = any (aval1%pdg == UNDEFINED) &
              .or. any (aval2%pdg == UNDEFINED)
         if (.not. eq) then
            allocate (match1 (size (aval1%pdg)))
            allocate (match2 (size (aval2%pdg)))
            match1 = .false.
            match2 = .false.
            do i = 1, size (aval1%pdg)
               match2 = match2 .or. aval1%pdg(i) == aval2%pdg
            end do
            do i = 1, size (aval2%pdg)
               match1 = match1 .or. aval2%pdg(i) == aval1%pdg
            end do
            eq = all (match1) .and. all (match2)
         end if
      else
         eq = .false.
      end if
    end function pdg_array_equivalent

    elemental function pdg_array_inequivalent (aval1, aval2) result (neq)
      logical :: neq
      type(pdg_array_t), intent(in) :: aval1, aval2
      neq = .not. pdg_array_equivalent (aval1, aval2)
    end function pdg_array_inequivalent
```

5.3.5 Sorting

Sort a PDG array by absolute value, particle before antiparticle. After sorting, we eliminate double entries.

⟨*PDG arrays: public*⟩+≡
```
    public :: sort_abs
```

⟨PDG arrays: interfaces⟩+≡
```
    interface sort_abs
       module procedure pdg_array_sort_abs
    end interface
```

⟨PDG arrays: pdg array: TBP⟩+≡
```
    procedure :: sort_abs => pdg_array_sort_abs
```
⟨PDG arrays: procedures⟩+≡
```
    function pdg_array_sort_abs (aval1, unique) result (aval2)
      class(pdg_array_t), intent(in) :: aval1
      logical, intent(in), optional :: unique
      type(pdg_array_t) :: aval2
      integer, dimension(:), allocatable :: tmp
      logical, dimension(:), allocatable :: mask
      integer :: i, n
      logical :: uni
      uni = .false.;  if (present (unique))  uni = unique
      n = size (aval1%pdg)
      if (uni) then
         allocate (tmp (n), mask(n))
         tmp = sort_abs (aval1%pdg)
         mask(1) = .true.
         do i = 2, n
            mask(i) = tmp(i) /= tmp(i-1)
         end do
         allocate (aval2%pdg (count (mask)))
         aval2%pdg = pack (tmp, mask)
      else
         allocate (aval2%pdg (n))
         aval2%pdg = sort_abs (aval1%pdg)
      end if
    end function pdg_array_sort_abs
```

5.3.6   PDG array list

A PDG array list, or PDG list, is an array of PDG-array objects with some
convenience methods.

⟨PDG arrays: public⟩+≡
```
    public :: pdg_list_t
```

⟨PDG arrays: types⟩+≡
```
    type :: pdg_list_t
      type(pdg_array_t), dimension(:), allocatable :: a
     contains
```
⟨PDG arrays: pdg list: TBP⟩
```
    end type pdg_list_t
```

Output, as a comma-separated list without advancing I/O.

⟨PDG arrays: pdg list: TBP⟩≡
```
    procedure :: write => pdg_list_write
```

⟨*PDG arrays: procedures*⟩+≡
```
    subroutine pdg_list_write (object, unit)
      class(pdg_list_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      if (allocated (object%a)) then
         do i = 1, size (object%a)
            if (i > 1)  write (u, "(A)", advance="no")  ", "
            call object%a(i)%write (u)
         end do
      end if
    end subroutine pdg_list_write
```

Initialize for a certain size. The entries are initially empty PDG arrays.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
    procedure :: init => pdg_list_init
```

⟨*PDG arrays: procedures*⟩+≡
```
    subroutine pdg_list_init (pl, n)
      class(pdg_list_t), intent(out) :: pl
      integer, intent(in) :: n
      allocate (pl%a (n))
    end subroutine pdg_list_init
```

Set one of the entries. No bounds-check.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
    generic :: set => pdg_list_set_int
    generic :: set => pdg_list_set_int_array
    generic :: set => pdg_list_set_pdg_array
    procedure, private :: pdg_list_set_int
    procedure, private :: pdg_list_set_int_array
    procedure, private :: pdg_list_set_pdg_array
```

⟨*PDG arrays: procedures*⟩+≡
```
    subroutine pdg_list_set_int (pl, i, pdg)
      class(pdg_list_t), intent(inout) :: pl
      integer, intent(in) :: i
      integer, intent(in) :: pdg
      pl%a(i) = pdg
    end subroutine pdg_list_set_int

    subroutine pdg_list_set_int_array (pl, i, pdg)
      class(pdg_list_t), intent(inout) :: pl
      integer, intent(in) :: i
      integer, dimension(:), intent(in) :: pdg
      pl%a(i) = pdg
    end subroutine pdg_list_set_int_array

    subroutine pdg_list_set_pdg_array (pl, i, pa)
      class(pdg_list_t), intent(inout) :: pl
      integer, intent(in) :: i
      type(pdg_array_t), intent(in) :: pa
      pl%a(i) = pa
```

```
      end subroutine pdg_list_set_pdg_array
```

Array size, not the length of individual entries

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
    procedure :: get_size => pdg_list_get_size
```

⟨*PDG arrays: procedures*⟩+≡
```
    function pdg_list_get_size (pl) result (n)
      class(pdg_list_t), intent(in) :: pl
      integer :: n
      if (allocated (pl%a)) then
         n = size (pl%a)
      else
         n = 0
      end if
    end function pdg_list_get_size
```

Return an entry, as a PDG array.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
    procedure :: get => pdg_list_get
```

⟨*PDG arrays: procedures*⟩+≡
```
    function pdg_list_get (pl, i) result (pa)
      class(pdg_list_t), intent(in) :: pl
      integer, intent(in) :: i
      type(pdg_array_t) :: pa
      pa = pl%a(i)
    end function pdg_list_get
```

Sort the list. First, each entry gets sorted, including elimination of doublers.
Then, we sort the list, using the first member of each PDG array as the marker.
No removal of doublers at this stage.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
    procedure :: sort_abs => pdg_list_sort_abs
```

⟨*PDG arrays: procedures*⟩+≡
```
    function pdg_list_sort_abs (pl) result (pl_sorted)
      class(pdg_list_t), intent(in) :: pl
      type(pdg_list_t) :: pl_sorted
      type(pdg_array_t), dimension(:), allocatable :: pa
      integer, dimension(:), allocatable :: pdg, map
      integer :: i
      call pl_sorted%init (pl%get_size ())
      if (allocated (pl%a)) then
         allocate (pa (size (pl%a)))
         do i = 1, size (pl%a)
            pa(i) = pl%a(i)%sort_abs (unique = .true.)
         end do
         allocate (pdg (size (pa)), source = 0)
         do i = 1, size (pa)
            if (allocated (pa(i)%pdg)) then
               if (size (pa(i)%pdg) > 0) then
                  pdg(i) = pa(i)%pdg(1)
```

374

```
              end if
           end if
        end do
        allocate (map (size (pdg)))
        map = order_abs (pdg)
        do i = 1, size (pa)
           call pl_sorted%set (i, pa(map(i)))
        end do
     end if
  end function pdg_list_sort_abs
```

Replace an entry. In the result, the entry `#i` is replaced by the contents of the second argument. The result is not sorted.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
    procedure :: replace => pdg_list_replace
```

⟨*PDG arrays: procedures*⟩+≡
```
    function pdg_list_replace (pl, i, pl_insert) result (pl_out)
      class(pdg_list_t), intent(in) :: pl
      integer, intent(in) :: i
      class(pdg_list_t), intent(in) :: pl_insert
      type(pdg_list_t) :: pl_out
      integer :: n, n_insert, n_out, k
      n = pl%get_size ()
      n_insert = pl_insert%get_size ()
      n_out = n + n_insert - 1
      call pl_out%init (n_out)
      if (allocated (pl%a)) then
         do k = 1, i - 1
            pl_out%a(k) = pl%a(k)
         end do
      end if
      if (allocated (pl_insert%a)) then
         do k = 1, n_insert
            pl_out%a(i-1+k) = pl_insert%a(k)
         end do
      end if
      if (allocated (pl%a)) then
         do k = 1, n - i
            pl_out%a(i+n_insert-1+k) = pl%a(i+k)
         end do
      end if
    end function pdg_list_replace
```

5.3.7 Unit tests

⟨*PDG arrays: public*⟩+≡
```
    public :: pdg_arrays_test
```

⟨*PDG arrays: procedures*⟩+≡
```
    subroutine pdg_arrays_test (u, results)
      integer, intent(in) :: u
      type (test_results_t), intent(inout) :: results
```

⟨*PDG arrays: execute tests*⟩
```
    end subroutine pdg_arrays_test
```

Basic functionality.

⟨*PDG arrays: execute tests*⟩≡
```
    call test (pdg_arrays_1, "pdg_arrays_1", &
         "create and sort PDG array", &
         u, results)
```

⟨*PDG arrays: tests*⟩≡
```
    subroutine pdg_arrays_1 (u)
      integer, intent(in) :: u

      type(pdg_array_t) :: pa, pa1, pa2, pa3, pa4, pa5, pa6
      integer, dimension(:), allocatable :: pdg

      write (u, "(A)")  "* Test output: pdg_arrays_1"
      write (u, "(A)")  "*   Purpose: create and sort PDG arrays"
      write (u, "(A)")

      write (u, "(A)")  "* Assignment"
      write (u, "(A)")

      call pa%write (u)
      write (u, *)
      write (u, "(A,I0)")  "length = ", pa%get_length ()
      pdg = pa
      write (u, "(A,3(1x,I0))")  "contents = ", pdg

      write (u, *)
      pa = 1
      call pa%write (u)
      write (u, *)
      write (u, "(A,I0)")  "length = ", pa%get_length ()
      pdg = pa
      write (u, "(A,3(1x,I0))")  "contents = ", pdg

      write (u, *)
      pa = [1, 2, 3]
      call pa%write (u)
      write (u, *)
      write (u, "(A,I0)")  "length = ", pa%get_length ()
      pdg = pa
      write (u, "(A,3(1x,I0))")  "contents = ", pdg
      write (u, "(A,I0)")  "element #2 = ", pa%get (2)

      write (u, *)
      write (u, "(A)")  "* Replace"
      write (u, *)

      pa = pa%replace (2, [-5, 5, -7])
      call pa%write (u)
      write (u, *)
```

```
write (u, *)
write (u, "(A)")  "* Sort"
write (u, *)

pa = [1, -7, 3, -5, 5, 3]
call pa%write (u)
write (u, *)
pa1 = pa%sort_abs ()
pa2 = pa%sort_abs (unique = .true.)
call pa1%write (u)
write (u, *)
call pa2%write (u)
write (u, *)

write (u, *)
write (u, "(A)")  "* Compare"
write (u, *)

pa1 = [1, 3]
pa2 = [1, 2, -2]
pa3 = [1, 2, 4]
pa4 = [1, 2, 4]
pa5 = [1, 2, -4]
pa6 = [1, 2, -3]

write (u, "(A,6(1x,L1))")  "< ", &
     pa1 < pa2, pa2 < pa3, pa3 < pa4, pa4 < pa5, pa5 < pa6, pa6 < pa1
write (u, "(A,6(1x,L1))")  "> ", &
     pa1 > pa2, pa2 > pa3, pa3 > pa4, pa4 > pa5, pa5 > pa6, pa6 > pa1
write (u, "(A,6(1x,L1))")  "<=", &
     pa1 <= pa2, pa2 <= pa3, pa3 <= pa4, pa4 <= pa5, pa5 <= pa6, pa6 <= pa1
write (u, "(A,6(1x,L1))")  ">=", &
     pa1 >= pa2, pa2 >= pa3, pa3 >= pa4, pa4 >= pa5, pa5 >= pa6, pa6 >= pa1
write (u, "(A,6(1x,L1))")  "==", &
     pa1 == pa2, pa2 == pa3, pa3 == pa4, pa4 == pa5, pa5 == pa6, pa6 == pa1
write (u, "(A,6(1x,L1))")  "/=", &
     pa1 /= pa2, pa2 /= pa3, pa3 /= pa4, pa4 /= pa5, pa5 /= pa6, pa6 /= pa1

write (u, *)
pa1 = [0]
pa2 = [1, 2]
pa3 = [1, -2]

write (u, "(A,6(1x,L1))")  "eqv ", &
     pa1 .eqv. pa1, pa1 .eqv. pa2, &
     pa2 .eqv. pa2, pa2 .eqv. pa3

write (u, "(A,6(1x,L1))")  "neqv", &
     pa1 .neqv. pa1, pa1 .neqv. pa2, &
     pa2 .neqv. pa2, pa2 .neqv. pa3


write (u, *)
write (u, "(A,6(1x,L1))")  "match", &
```

377

```
            pa1 .match. 0, pa1 .match. 1, &
            pa2 .match. 0, pa2 .match. 1, pa2 .match. 3

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: pdg_arrays_1"

    end subroutine pdg_arrays_1
```

PDG array list, i.e., arrays of arrays.

⟨*PDG arrays: execute tests*⟩+≡
```
    call test (pdg_arrays_2, "pdg_arrays_2", &
         "create and sort PDG array", &
         u, results)
```
⟨*PDG arrays: tests*⟩+≡
```
    subroutine pdg_arrays_2 (u)
      integer, intent(in) :: u

      type(pdg_array_t) :: pa
      type(pdg_list_t) :: pl, pl1

      write (u, "(A)")  "* Test output: pdg_arrays_2"
      write (u, "(A)")  "*   Purpose: create and sort PDG lists"
      write (u, "(A)")

      write (u, "(A)")  "* Assignment"
      write (u, "(A)")

      call pl%init (3)
      call pl%set (1, 42)
      call pl%set (2, [3, 2])
      pa = [5, -5]
      call pl%set (3, pa)
      call pl%write (u)
      write (u, *)
      write (u, "(A,I0)")  "size = ", pl%get_size ()

      write (u, "(A)")
      write (u, "(A)")  "* Sort"
      write (u, "(A)")

      pl = pl%sort_abs ()
      call pl%write (u)
      write (u, *)

      write (u, "(A)")
      write (u, "(A)")  "* Extract item #3"
      write (u, "(A)")

      pa = pl%get (3)
      call pa%write (u)
      write (u, *)

      write (u, "(A)")
```

```
      write (u, "(A)")  "* Replace item #3"
      write (u, "(A)")

      call pl1%init (2)
      call pl1%set (1, [2, 4])
      call pl1%set (2, -7)

      pl = pl%replace (3, pl1)
      call pl%write (u)
      write (u, *)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: pdg_arrays_2"

    end subroutine pdg_arrays_2
```

5.4 subevents

The purpose of subevents is to store the relevant part of the physical event (either partonic or hadronic), and to hold particle selections and combinations which are constructed in cut or analysis expressions.

⟨subevents.f90⟩≡
  ⟨*File header*⟩

```
  module subevents

    use iso_c_binding !NODEP!
```
⟨*Use kinds*⟩
⟨*Use file utils*⟩
```
    use limits, only: FMT_14, FMT_19 !NODEP!
    use c_particles !NODEP!
    use jets
    use lorentz !NODEP!
    use sorting
    use pdg_arrays
```

⟨*Standard module head*⟩

⟨*Subevents: public*⟩

⟨*Subevents: parameters*⟩

⟨*Subevents: types*⟩

⟨*Subevents: interfaces*⟩

```
  contains
```

⟨*Subevents: procedures*⟩

```
  end module subevents
```

5.4.1 Particles

For the purpose of this module, a particle has a type which can indicate a beam, incoming, outgoing, or composite particle, flavor and helicity codes (integer, undefined for composite), four-momentum and invariant mass squared. (Other particles types are used in extended event types, but also defined here.) Furthermore, each particle has an allocatable array of ancestors – particle indices which indicate the building blocks of a composite particle. For an incoming/outgoing particle, the array contains only the index of the particle itself.

For incoming particles, the momentum is inverted before storing it in the particle object.

⟨*Subevents: parameters*⟩≡
```
    integer, parameter, public :: PRT_UNDEFINED = 0
    integer, parameter, public :: PRT_BEAM = -9
    integer, parameter, public :: PRT_INCOMING = 1
    integer, parameter, public :: PRT_OUTGOING = 2
```

```
integer, parameter, public :: PRT_COMPOSITE = 3
integer, parameter, public :: PRT_VIRTUAL = 4
integer, parameter, public :: PRT_RESONANT = 5
integer, parameter, public :: PRT_BEAM_REMNANT = 9
```

The type

We initialize only the type here and mark as unpolarized. The initializers below
do the rest.

⟨*Subevents: public*⟩≡
```
public :: prt_t
```

⟨*Subevents: types*⟩≡
```
type :: prt_t
   private
   integer :: type = PRT_UNDEFINED
   integer :: pdg
   logical :: polarized = .false.
   integer :: h
   type(vector4_t) :: p
   real(default) :: p2
   integer, dimension(:), allocatable :: src
end type prt_t
```

Initializers. Polarization is set separately. Finalizers are not needed.

⟨*Subevents: procedures*⟩≡
```
subroutine prt_init_beam (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_BEAM
  call prt_set (prt, pdg, - p, p2, src)
end subroutine prt_init_beam

subroutine prt_init_incoming (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_INCOMING
  call prt_set (prt, pdg, - p, p2, src)
end subroutine prt_init_incoming

subroutine prt_init_outgoing (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
```

```
      prt%type = PRT_OUTGOING
      call prt_set (prt, pdg, p, p2, src)
    end subroutine prt_init_outgoing

    subroutine prt_init_composite (prt, p, src)
      type(prt_t), intent(out) :: prt
      type(vector4_t), intent(in) :: p
      integer, dimension(:), intent(in) :: src
      prt%type = PRT_COMPOSITE
      call prt_set (prt, 0, p, p**2, src)
    end subroutine prt_init_composite
```

This version is for temporary particle objects, so the `src` array is not set.

⟨*Subevents: public*⟩+≡
```
    public :: prt_init_combine
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine prt_init_combine (prt, prt1, prt2)
      type(prt_t), intent(out) :: prt
      type(prt_t), intent(in) :: prt1, prt2
      type(vector4_t) :: p
      integer, dimension(0) :: src
      prt%type = PRT_COMPOSITE
      p = prt1%p + prt2%p
      call prt_set (prt, 0, p, p**2, src)
    end subroutine prt_init_combine
```

Init from a pseudojet object.

⟨*Subevents: procedures*⟩+≡
```
    subroutine prt_init_pseudojet (prt, jet, src)
      type(prt_t), intent(out) :: prt
      type(pseudojet_t), intent(in) :: jet
      integer, dimension(:), intent(in) :: src
      type(vector4_t) :: p
      prt%type = PRT_COMPOSITE
      p = vector4_moving (jet%e(), &
          vector3_moving ([jet%px(), jet%py(), jet%pz()]))
      call prt_set (prt, 0, p, p**2, src)
    end subroutine prt_init_pseudojet
```

Accessing contents

⟨*Subevents: public*⟩+≡
```
    public :: prt_get_pdg
```

⟨*Subevents: procedures*⟩+≡
```
    elemental function prt_get_pdg (prt) result (pdg)
      integer :: pdg
      type(prt_t), intent(in) :: prt
      pdg = prt%pdg
    end function prt_get_pdg
```

⟨*Subevents: public*⟩+≡
```
    public :: prt_get_momentum
```

⟨*Subevents: procedures*⟩+≡
```
    elemental function prt_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(prt_t), intent(in) :: prt
      p = prt%p
    end function prt_get_momentum
```

⟨*Subevents: public*⟩+≡
```
    public :: prt_get_msq
```

⟨*Subevents: procedures*⟩+≡
```
    elemental function prt_get_msq (prt) result (msq)
      real(default) :: msq
      type(prt_t), intent(in) :: prt
      msq = prt%p2
    end function prt_get_msq
```

⟨*Subevents: public*⟩+≡
```
    public :: prt_is_polarized
```

⟨*Subevents: procedures*⟩+≡
```
    elemental function prt_is_polarized (prt) result (flag)
      logical :: flag
      type(prt_t), intent(in) :: prt
      flag = prt%polarized
    end function prt_is_polarized
```

⟨*Subevents: public*⟩+≡
```
    public :: prt_get_helicity
```

⟨*Subevents: procedures*⟩+≡
```
    elemental function prt_get_helicity (prt) result (h)
      integer :: h
      type(prt_t), intent(in) :: prt
      h = prt%h
    end function prt_get_helicity
```

Setting data

Set the PDG, momentum and momentum squared, and ancestors. If allocate-on-assignment is available, this can be simplified.

⟨*Subevents: procedures*⟩+≡
```
    subroutine prt_set (prt, pdg, p, p2, src)
      type(prt_t), intent(inout) :: prt
      integer, intent(in) :: pdg
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: p2
      integer, dimension(:), intent(in) :: src
      prt%pdg = pdg
      prt%p = p
```

```
      prt%p2 = p2
      if (allocated (prt%src)) then
         if (size (src) /= size (prt%src)) then
            deallocate (prt%src)
            allocate (prt%src (size (src)))
         end if
      else
         allocate (prt%src (size (src)))
      end if
      prt%src = src
    end subroutine prt_set
```

Set the particle PDG code separately.

⟨*Subevents: procedures*⟩+≡
```
    elemental subroutine prt_set_pdg (prt, pdg)
      type(prt_t), intent(inout) :: prt
      integer, intent(in) :: pdg
      prt%pdg = pdg
    end subroutine prt_set_pdg
```

Set the momentum separately.

⟨*Subevents: procedures*⟩+≡
```
    elemental subroutine prt_set_p (prt, p)
      type(prt_t), intent(inout) :: prt
      type(vector4_t), intent(in) :: p
      prt%p = p
    end subroutine prt_set_p
```

Set the squared invariant mass separately.

⟨*Subevents: procedures*⟩+≡
```
    elemental subroutine prt_set_p2 (prt, p2)
      type(prt_t), intent(inout) :: prt
      real(default), intent(in) :: p2
      prt%p2 = p2
    end subroutine prt_set_p2
```

Set helicity (optional).

⟨*Subevents: procedures*⟩+≡
```
    subroutine prt_polarize (prt, h)
      type(prt_t), intent(inout) :: prt
      integer, intent(in) :: h
      prt%polarized = .true.
      prt%h = h
    end subroutine prt_polarize
```

Conversion

Transform a `prt_t` object into a `c_prt_t` object.

⟨*Subevents: public*⟩+≡
```
    public :: c_prt
```

⟨*Subevents: interfaces*⟩≡
```
    interface c_prt
       module procedure c_prt_from_prt
    end interface
```

⟨*Subevents: procedures*⟩+≡
```
    elemental function c_prt_from_prt (prt) result (c_prt)
      type(c_prt_t) :: c_prt
      type(prt_t), intent(in) :: prt
      c_prt%type = prt%type
      c_prt%pdg = prt%pdg
      if (prt%polarized) then
         c_prt%polarized = 1
      else
         c_prt%polarized = 0
      end if
      c_prt%h = prt%h
      c_prt%pe = energy (prt%p)
      c_prt%px = vector4_get_component (prt%p, 1)
      c_prt%py = vector4_get_component (prt%p, 2)
      c_prt%pz = vector4_get_component (prt%p, 3)
      c_prt%p2 = prt%p2
    end function c_prt_from_prt
```

Output

⟨*Subevents: public*⟩+≡
```
    public :: prt_write
```
⟨*Subevents: procedures*⟩+≡
```
    subroutine prt_write (prt, unit)
      type(prt_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)", advance="no")  "prt("
      select case (prt%type)
      case (PRT_UNDEFINED);    write (u, "('?')", advance="no")
      case (PRT_BEAM);         write (u, "('b:')", advance="no")
      case (PRT_INCOMING);     write (u, "('i:')", advance="no")
      case (PRT_OUTGOING);     write (u, "('o:')", advance="no")
      case (PRT_COMPOSITE);    write (u, "('c:')", advance="no")
      end select
      select case (prt%type)
      case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
         if (prt%polarized) then
            write (u, "(I0,'/',I0,'|')", advance="no")  prt%pdg, prt%h
         else
            write (u, "(I0,'|')", advance="no") prt%pdg
         end if
      end select
      select case (prt%type)
      case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING, PRT_COMPOSITE)
```

385

```
        write (u, "(" // FMT_14 // ",';'," // FMT_14 // ",','," // &
             FMT_14 // ",','," // FMT_14 // ")", advance="no") &
             array_from_vector4 (prt%p)
        write (u, "('|'," // FMT_19 // ")", advance="no")  prt%p2
     end select
     if (allocated (prt%src)) then
        write (u, "('|')", advance="no")
        do i = 1, size (prt%src)
           write (u, "(1x,I0)", advance="no")  prt%src(i)
        end do
     end if
     write (u, "(A)")  ")"
   end subroutine prt_write
```

Tools

Two particles match if their `src` arrays are the same.

⟨*Subevents: interfaces*⟩+≡
```
   interface operator(.match.)
      module procedure prt_match
   end interface
```

⟨*Subevents: procedures*⟩+≡
```
   elemental function prt_match (prt1, prt2) result (match)
     logical :: match
     type(prt_t), intent(in) :: prt1, prt2
     if (size (prt1%src) == size (prt2%src)) then
        match = all (prt1%src == prt2%src)
     else
        match = .false.
     end if
   end function prt_match
```

The combine operation makes a pseudoparticle whose momentum is the result of adding (the momenta of) the pair of input particles. We trace the particles from which a particle is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

The result is intent(inout), so it will not be initialized when the subroutine is entered.

⟨*Subevents: procedures*⟩+≡
```
   subroutine prt_combine (prt, prt_in1, prt_in2, ok)
     type(prt_t), intent(inout) :: prt
     type(prt_t), intent(in) :: prt_in1, prt_in2
     logical :: ok
     integer, dimension(:), allocatable :: src
     call combine_index_lists (src, prt_in1%src, prt_in2%src)
```

```
      ok = allocated (src)
      if (ok)  call prt_init_composite (prt, prt_in1%p + prt_in2%p, src)
   end subroutine prt_combine
```

This variant does not produce the combined particle, it just checks whether the combination is valid (no common `src` entry).

⟨*Subevents: public*⟩+≡
```
   public :: are_disjoint
```

⟨*Subevents: procedures*⟩+≡
```
   function are_disjoint (prt_in1, prt_in2) result (flag)
     logical :: flag
     type(prt_t), intent(in) :: prt_in1, prt_in2
     flag = index_lists_are_disjoint (prt_in1%src, prt_in2%src)
   end function are_disjoint
```

`src` Lists with length > 1 are built by a `combine` operation which merges the lists in a sorted manner. If the result would have a duplicate entry, it is discarded, and the result is unallocated.

⟨*Subevents: procedures*⟩+≡
```
   subroutine combine_index_lists (res, src1, src2)
     integer, dimension(:), intent(in) :: src1, src2
     integer, dimension(:), allocatable :: res
     integer :: i1, i2, i
     allocate (res (size (src1) + size (src2)))
     if (size (src1) == 0) then
        res = src2
        return
     else if (size (src2) == 0) then
        res = src1
        return
     end if
     i1 = 1
     i2 = 1
     LOOP: do i = 1, size (res)
        if (src1(i1) < src2(i2)) then
           res(i) = src1(i1);   i1 = i1 + 1
           if (i1 > size (src1)) then
              res(i+1:) = src2(i2:)
              exit LOOP
           end if
        else if (src1(i1) > src2(i2)) then
           res(i) = src2(i2);   i2 = i2 + 1
           if (i2 > size (src2)) then
              res(i+1:) = src1(i1:)
              exit LOOP
           end if
        else
           deallocate (res)
           exit LOOP
        end if
     end do LOOP
   end subroutine combine_index_lists
```

This function is similar, but it does not actually merge the list, it just checks whether they are disjoint (no common `src` entry).

⟨*Subevents: procedures*⟩+≡
```
function index_lists_are_disjoint (src1, src2) result (flag)
  logical :: flag
  integer, dimension(:), intent(in) :: src1, src2
  integer :: i1, i2, i
  flag = .true.
  i1 = 1
  i2 = 1
  LOOP: do i = 1, size (src1) + size (src2)
     if (src1(i1) < src2(i2)) then
        i1 = i1 + 1
        if (i1 > size (src1)) then
           exit LOOP
        end if
     else if (src1(i1) > src2(i2)) then
        i2 = i2 + 1
        if (i2 > size (src2)) then
           exit LOOP
        end if
     else
        flag = .false.
        exit LOOP
     end if
  end do LOOP
end function index_lists_are_disjoint
```

5.4.2 C-compatible particle type

5.4.3 subevents

Particles are collected in subevents. This type is implemented as a dynamically allocated array, which need not be completely filled. The value `n_active` determines the number of meaningful entries.

Type definition

⟨*Subevents: public*⟩+≡
```
public :: subevt_t
```

⟨*Subevents: types*⟩+≡
```
type :: subevt_t
  private
  integer :: n_active = 0
  type(prt_t), dimension(:), allocatable :: prt
 contains
 ⟨Subevents: subevt: TBP⟩
end type subevt_t
```

Initialize, allocating with size zero (default) or given size. The number of contained particles is set equal to the size.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_init
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_init (subevt, n_active)
      type(subevt_t), intent(out) :: subevt
      integer, intent(in), optional :: n_active
      if (present (n_active))  subevt%n_active = n_active
      allocate (subevt%prt (subevt%n_active))
    end subroutine subevt_init
```

(Re-)allocate the subevent with some given size. If the size is greater than the previous one, do a real reallocation. Otherwise, just reset the recorded size. Contents are untouched, but become invalid.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_reset
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_reset (subevt, n_active)
      type(subevt_t), intent(inout) :: subevt
      integer, intent(in) :: n_active
      subevt%n_active = n_active
      if (subevt%n_active > size (subevt%prt)) then
         deallocate (subevt%prt)
         allocate (subevt%prt (subevt%n_active))
      end if
    end subroutine subevt_reset
```

Output. No prefix for the headline 'subevt', because this will usually be printed appending to a previous line.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_write
```

⟨*Subevents: subevt: TBP*⟩≡
```
    procedure :: write => subevt_write
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_write (object, unit, prefix)
      class(subevt_t), intent(in) :: object
      integer, intent(in), optional :: unit
      character(*), intent(in), optional :: prefix
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)") "subevent:"
      do i = 1, object%n_active
         if (present (prefix))  write (u, "(A)", advance="no") prefix
         write (u, "(1x,I0)", advance="no")  i
         call prt_write (object%prt(i), unit)
      end do
    end subroutine subevt_write
```

Defined assignment: transfer only meaningful entries. This is a deep copy (as would be default assignment).

⟨*Subevents: interfaces*⟩+≡
```
interface assignment(=)
   module procedure subevt_assign
end interface
```

⟨*Subevents: procedures*⟩+≡
```
subroutine subevt_assign (subevt, subevt_in)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: subevt_in
  if (.not. allocated (subevt%prt)) then
     call subevt_init (subevt, subevt_in%n_active)
  else
     call subevt_reset (subevt, subevt_in%n_active)
  end if
  subevt%prt(:subevt%n_active) = subevt_in%prt(:subevt%n_active)
end subroutine subevt_assign
```

Fill contents

Store incoming/outgoing particles which are completely defined.

⟨*Subevents: public*⟩+≡
```
public :: subevt_set_beam
public :: subevt_set_incoming
public :: subevt_set_outgoing
public :: subevt_set_composite
```

⟨*Subevents: procedures*⟩+≡
```
subroutine subevt_set_beam (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
     call prt_init_beam (subevt%prt(i), pdg, p, p2, src)
  else
     call prt_init_beam (subevt%prt(i), pdg, p, p2, (/ i /))
  end if
end subroutine subevt_set_beam

subroutine subevt_set_incoming (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
     call prt_init_incoming (subevt%prt(i), pdg, p, p2, src)
```

```
      else
         call prt_init_incoming (subevt%prt(i), pdg, p, p2, (/ i /))
      end if
   end subroutine subevt_set_incoming

   subroutine subevt_set_outgoing (subevt, i, pdg, p, p2, src)
     type(subevt_t), intent(inout) :: subevt
     integer, intent(in) :: i
     integer, intent(in) :: pdg
     type(vector4_t), intent(in) :: p
     real(default), intent(in) :: p2
     integer, dimension(:), intent(in), optional :: src
     if (present (src)) then
        call prt_init_outgoing (subevt%prt(i), pdg, p, p2, src)
     else
        call prt_init_outgoing (subevt%prt(i), pdg, p, p2, (/ i /))
     end if
   end subroutine subevt_set_outgoing

   subroutine subevt_set_composite (subevt, i, p, src)
     type(subevt_t), intent(inout) :: subevt
     integer, intent(in) :: i
     type(vector4_t), intent(in) :: p
     integer, dimension(:), intent(in) :: src
     call prt_init_composite (subevt%prt(i), p, src)
   end subroutine subevt_set_composite
```

Separately assign flavors, simultaneously for all incoming/outgoing particles.

⟨*Subevents: public*⟩+≡
```
   public :: subevt_set_pdg_beam
   public :: subevt_set_pdg_incoming
   public :: subevt_set_pdg_outgoing
```

⟨*Subevents: procedures*⟩+≡
```
   subroutine subevt_set_pdg_beam (subevt, pdg)
     type(subevt_t), intent(inout) :: subevt
     integer, dimension(:), intent(in) :: pdg
     integer :: i, j
     j = 1
     do i = 1, subevt%n_active
        if (subevt%prt(i)%type == PRT_BEAM) then
           call prt_set_pdg (subevt%prt(i), pdg(j))
           j = j + 1
           if (j > size (pdg))  exit
        end if
     end do
   end subroutine subevt_set_pdg_beam

   subroutine subevt_set_pdg_incoming (subevt, pdg)
     type(subevt_t), intent(inout) :: subevt
     integer, dimension(:), intent(in) :: pdg
     integer :: i, j
     j = 1
     do i = 1, subevt%n_active
```

```
      if (subevt%prt(i)%type == PRT_INCOMING) then
         call prt_set_pdg (subevt%prt(i), pdg(j))
         j = j + 1
         if (j > size (pdg))  exit
      end if
   end do
end subroutine subevt_set_pdg_incoming

subroutine subevt_set_pdg_outgoing (subevt, pdg)
  type(subevt_t), intent(inout) :: subevt
  integer, dimension(:), intent(in) :: pdg
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_OUTGOING) then
        call prt_set_pdg (subevt%prt(i), pdg(j))
        j = j + 1
        if (j > size (pdg))  exit
     end if
  end do
end subroutine subevt_set_pdg_outgoing
```

Separately assign momenta, simultaneously for all incoming/outgoing particles.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_set_p_beam
    public :: subevt_set_p_incoming
    public :: subevt_set_p_outgoing
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_set_p_beam (subevt, p)
      type(subevt_t), intent(inout) :: subevt
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: i, j
      j = 1
      do i = 1, subevt%n_active
         if (subevt%prt(i)%type == PRT_BEAM) then
            call prt_set_p (subevt%prt(i), p(j))
            j = j + 1
            if (j > size (p))  exit
         end if
      end do
    end subroutine subevt_set_p_beam

    subroutine subevt_set_p_incoming (subevt, p)
      type(subevt_t), intent(inout) :: subevt
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: i, j
      j = 1
      do i = 1, subevt%n_active
         if (subevt%prt(i)%type == PRT_INCOMING) then
            call prt_set_p (subevt%prt(i), p(j))
            j = j + 1
            if (j > size (p))  exit
         end if
```

392

```
      end do
    end subroutine subevt_set_p_incoming

    subroutine subevt_set_p_outgoing (subevt, p)
      type(subevt_t), intent(inout) :: subevt
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: i, j
      j = 1
      do i = 1, subevt%n_active
         if (subevt%prt(i)%type == PRT_OUTGOING) then
            call prt_set_p (subevt%prt(i), p(j))
            j = j + 1
            if (j > size (p))  exit
         end if
      end do
    end subroutine subevt_set_p_outgoing
```

Separately assign the squared invariant mass, simultaneously for all incoming/outgoing particles.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_set_p2_beam
    public :: subevt_set_p2_incoming
    public :: subevt_set_p2_outgoing
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_set_p2_beam (subevt, p2)
      type(subevt_t), intent(inout) :: subevt
      real(default), dimension(:), intent(in) :: p2
      integer :: i, j
      j = 1
      do i = 1, subevt%n_active
         if (subevt%prt(i)%type == PRT_BEAM) then
            call prt_set_p2 (subevt%prt(i), p2(j))
            j = j + 1
            if (j > size (p2))  exit
         end if
      end do
    end subroutine subevt_set_p2_beam

    subroutine subevt_set_p2_incoming (subevt, p2)
      type(subevt_t), intent(inout) :: subevt
      real(default), dimension(:), intent(in) :: p2
      integer :: i, j
      j = 1
      do i = 1, subevt%n_active
         if (subevt%prt(i)%type == PRT_INCOMING) then
            call prt_set_p2 (subevt%prt(i), p2(j))
            j = j + 1
            if (j > size (p2))  exit
         end if
      end do
    end subroutine subevt_set_p2_incoming

    subroutine subevt_set_p2_outgoing (subevt, p2)
```

```
      type(subevt_t), intent(inout) :: subevt
      real(default), dimension(:), intent(in) :: p2
      integer :: i, j
      j = 1
      do i = 1, subevt%n_active
         if (subevt%prt(i)%type == PRT_OUTGOING) then
            call prt_set_p2 (subevt%prt(i), p2(j))
            j = j + 1
            if (j > size (p2))  exit
         end if
      end do
   end subroutine subevt_set_p2_outgoing
```

Set polarization for an entry

⟨*Subevents: public*⟩+≡
```
    public :: subevt_polarize
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_polarize (subevt, i, h)
      type(subevt_t), intent(inout) :: subevt
      integer, intent(in) :: i, h
      call prt_polarize (subevt%prt(i), h)
    end subroutine subevt_polarize
```

Accessing contents

Return true if the subevent has entries.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_is_nonempty
```

⟨*Subevents: procedures*⟩+≡
```
    function subevt_is_nonempty (subevt) result (flag)
      logical :: flag
      type(subevt_t), intent(in) :: subevt
      flag = subevt%n_active /= 0
    end function subevt_is_nonempty
```

Return the number of entries

⟨*Subevents: public*⟩+≡
```
    public :: subevt_get_length
```

⟨*Subevents: procedures*⟩+≡
```
    function subevt_get_length (subevt) result (length)
      integer :: length
      type(subevt_t), intent(in) :: subevt
      length = subevt%n_active
    end function subevt_get_length
```

Return a specific particle. The index is not checked for validity.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_get_prt
```

⟨*Subevents: procedures*⟩+≡
```
function subevt_get_prt (subevt, i) result (prt)
  type(prt_t) :: prt
  type(subevt_t), intent(in) :: subevt
  integer, intent(in) :: i
  prt = subevt%prt(i)
end function subevt_get_prt
```

Return the partonic energy squared. We take the particles with flag `PRT_INCOMING` and compute their total invariant mass.

⟨*Subevents: public*⟩+≡
```
public :: subevt_get_sqrts_hat
```

⟨*Subevents: procedures*⟩+≡
```
function subevt_get_sqrts_hat (subevt) result (sqrts_hat)
  type(subevt_t), intent(in) :: subevt
  real(default) :: sqrts_hat
  type(vector4_t) :: p
  integer :: i
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_INCOMING) then
        p = p + prt_get_momentum (subevt%prt(i))
     end if
  end do
  sqrts_hat = p ** 1
end function subevt_get_sqrts_hat
```

⟨*Subevents: interfaces*⟩+≡
```
interface c_prt
  module procedure c_prt_from_subevt
  module procedure c_prt_array_from_subevt
end interface
```

⟨*Subevents: procedures*⟩+≡
```
function c_prt_from_subevt (subevt, i) result (c_prt)
  type(c_prt_t) :: c_prt
  type(subevt_t), intent(in) :: subevt
  integer, intent(in) :: i
  c_prt = c_prt_from_prt (subevt%prt(i))
end function c_prt_from_subevt

function c_prt_array_from_subevt (subevt) result (c_prt_array)
  type(subevt_t), intent(in) :: subevt
  type(c_prt_t), dimension(subevt%n_active) :: c_prt_array
  c_prt_array = c_prt_from_prt (subevt%prt(1:subevt%n_active))
end function c_prt_array_from_subevt
```

Operations with subevents

The join operation joins two subevents. When appending the elements of the second list, we check for each particle whether it is already in the first list. If yes, it is discarded. The result list should be initialized already.

If a mask is present, it refers to the second subevent. Particles where the mask is not set are discarded.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_join
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_join (subevt, pl1, pl2, mask2)
      type(subevt_t), intent(inout) :: subevt
      type(subevt_t), intent(in) :: pl1, pl2
      logical, dimension(:), intent(in), optional :: mask2
      integer :: n1, n2, i, n
      n1 = pl1%n_active
      n2 = pl2%n_active
      call subevt_reset (subevt, n1 + n2)
      subevt%prt(:n1)   = pl1%prt(:n1)
      n = n1
      if (present (mask2)) then
         do i = 1, pl2%n_active
            if (mask2(i)) then
               if (disjoint (i)) then
                  n = n + 1
                  subevt%prt(n) = pl2%prt(i)
               end if
            end if
         end do
      else
         do i = 1, pl2%n_active
            if (disjoint (i)) then
               n = n + 1
               subevt%prt(n) = pl2%prt(i)
            end if
         end do
      end if
      subevt%n_active = n
    contains
      function disjoint (i) result (flag)
        integer, intent(in) :: i
        logical :: flag
        integer :: j
        do j = 1, pl1%n_active
           if (.not. are_disjoint (pl1%prt(j), pl2%prt(i))) then
              flag = .false.
              return
           end if
        end do
        flag = .true.
      end function disjoint
    end subroutine subevt_join
```

The combine operation makes a subevent whose entries are the result of adding (the momenta of) each pair of particles in the input lists. We trace the particles from which a particles is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices

refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_combine
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_combine (subevt, pl1, pl2, mask12)
      type(subevt_t), intent(inout) :: subevt
      type(subevt_t), intent(in) :: pl1, pl2
      logical, dimension(:,:), intent(in), optional :: mask12
      integer :: n1, n2, i1, i2, n, j
      logical :: ok
      n1 = pl1%n_active
      n2 = pl2%n_active
      call subevt_reset (subevt, n1 * n2)
      n = 1
      do i1 = 1, n1
         do i2 = 1, n2
            if (present (mask12)) then
               ok = mask12(i1,i2)
            else
               ok = .true.
            end if
            if (ok)  call prt_combine &
                 (subevt%prt(n), pl1%prt(i1), pl2%prt(i2), ok)
            if (ok) then
               CHECK_DOUBLES: do j = 1, n - 1
                  if (subevt%prt(n) .match. subevt%prt(j)) then
                     ok = .false.;  exit CHECK_DOUBLES
                  end if
               end do CHECK_DOUBLES
               if (ok)  n = n + 1
            end if
         end do
      end do
      subevt%n_active = n - 1
    end subroutine subevt_combine
```

The collect operation makes a single-entry subevent which results from combining (the momenta of) all particles in the input list. As above, the result does not contain an original particle more than once; this is checked for each particle when it is collected. Furthermore, each entry has a mask; where the mask is false, the entry is dropped.

(Thus, if the input particles are already composite, there is some chance that the result depends on the order of the input list and is not as expected. This situation should be avoided.)

⟨*Subevents: public*⟩+≡
```
    public :: subevt_collect
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_collect (subevt, pl1, mask1)
```

```
type(subevt_t), intent(inout) :: subevt
type(subevt_t), intent(in) :: pl1
logical, dimension(:), intent(in) :: mask1
type(prt_t) :: prt
integer :: i
logical :: ok
call subevt_reset (subevt, 1)
subevt%n_active = 0
do i = 1, pl1%n_active
   if (mask1(i)) then
      if (subevt%n_active == 0) then
         subevt%n_active = 1
         subevt%prt(1) = pl1%prt(i)
      else
         call prt_combine (prt, subevt%prt(1), pl1%prt(i), ok)
         if (ok)  subevt%prt(1) = prt
      end if
   end if
end do
end subroutine subevt_collect
```

The cluster operation is similar to `collect`, but applies a jet algorithm. The result is a subevent consisting of jets and, possibly, unclustered extra particles. As above, the result does not contain an original particle more than once; this is checked for each particle when it is collected. Furthermore, each entry has a mask; where the mask is false, the entry is dropped.

The algorithm: first determine the (pseudo)particles that participate in the clustering. They should not overlap, and the mask entry must be set. We then cluster the particles, using the given jet definition. The result particles are retrieved from the cluster sequence. We still have to determine the source indices for each jet: for each input particle, we get the jet index. Accumulating the source entries for all particles that are part of a given jet, we derive the jet source entries. Finally, we delete the C structures that have been constructed by FastJet and its interface.

⟨*Subevents: public*⟩+≡
```
public :: subevt_cluster
```

⟨*Subevents: procedures*⟩+≡
```
subroutine subevt_cluster (subevt, pl1, mask1, jet_def)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl1
  logical, dimension(:), intent(in) :: mask1
  type(jet_definition_t), intent(in) :: jet_def
  integer, dimension(:), allocatable :: src, src_tmp
  integer, dimension(:), allocatable :: map, jet_idx
  type(pseudojet_t), dimension(:), allocatable :: jet_in, jet_out
  type(pseudojet_vector_t) :: jv_in, jv_out
  type(cluster_sequence_t) :: cs
  integer :: i, j, k, n_src, n_active
  n_active = 0
  allocate (map (pl1%n_active), source = 0)
  allocate (src (0))
  do i = 1, pl1%n_active
```

```
        if (mask1(i)) then
            call combine_index_lists (src_tmp, src, pl1%prt(i)%src)
            if (allocated (src_tmp)) then
                call move_alloc (from=src_tmp, to=src)
                n_active = n_active + 1
                map(n_active) = i
            end if
        end if
    end do
    allocate (jet_in (count (map /= 0)))
    do i = 1, size (jet_in)
        call jet_in(i)%init (prt_get_momentum (pl1%prt(map(i))))
    end do
    call jv_in%init (jet_in)
    call cs%init (jv_in, jet_def)
    jv_out = cs%inclusive_jets ()
    allocate (jet_idx (size (jet_in)))
    call cs%assign_jet_indices (jv_out, jet_idx)
    allocate (jet_out (jv_out%size ()))
    jet_out = jv_out
    call subevt_reset (subevt, size (jet_out))
    do i = 1, size (jet_out)
        src = 0
        n_src = 0
        do j = 1, size (jet_idx)
            if (jet_idx(j) == i) then
                associate (prt => pl1%prt(map(j)))
                    do k = 1, size (prt%src)
                        src(n_src + k) = prt%src(k)
                    end do
                    n_src = n_src + size (prt%src)
                end associate
            end if
        end do
        call prt_init_pseudojet (subevt%prt(i), jet_out(i), src(:n_src))
    end do
    do i = 1, size (jet_out)
        call jet_out(i)%final ()
    end do
    call jv_out%final ()
    call cs%final ()
    call jv_in%final ()
    do i = 1, size (jet_in)
        call jet_in(i)%final ()
    end do
    end subroutine subevt_cluster
```

Return a list of all particles for which the mask is true.

⟨Subevents: public⟩+≡
```
    public :: subevt_select
```

⟨Subevents: procedures⟩+≡
```
    subroutine subevt_select (subevt, pl, mask1)
      type(subevt_t), intent(inout) :: subevt
```

```
        type(subevt_t), intent(in) :: pl
        logical, dimension(:), intent(in) :: mask1
        integer :: i, n
        call subevt_reset (subevt, pl%n_active)
        n = 0
        do i = 1, pl%n_active
           if (mask1(i)) then
              n = n + 1
              subevt%prt(n) = pl%prt(i)
           end if
        end do
        subevt%n_active = n
     end subroutine subevt_select
```

Return a subevent which consists of the single particle with specified `index`. If `index` is negative, count from the end. If it is out of bounds, return an empty list.

⟨*Subevents: public*⟩+≡
```
     public :: subevt_extract
```

⟨*Subevents: procedures*⟩+≡
```
     subroutine subevt_extract (subevt, pl, index)
       type(subevt_t), intent(inout) :: subevt
       type(subevt_t), intent(in) :: pl
       integer, intent(in) :: index
       if (index > 0) then
          if (index <= pl%n_active) then
             call subevt_reset (subevt, 1)
             subevt%prt(1) = pl%prt(index)
          else
             call subevt_reset (subevt, 0)
          end if
       else if (index < 0) then
          if (abs (index) <= pl%n_active) then
             call subevt_reset (subevt, 1)
             subevt%prt(1) = pl%prt(pl%n_active + 1 + index)
          else
             call subevt_reset (subevt, 0)
          end if
       else
          call subevt_reset (subevt, 0)
       end if
     end subroutine subevt_extract
```

Return the list of particles sorted according to increasing values of the provided integer or real array. If no array is given, sort by PDG value.

⟨*Subevents: public*⟩+≡
```
     public :: subevt_sort
```

⟨*Subevents: interfaces*⟩+≡
```
     interface subevt_sort
        module procedure subevt_sort_pdg
        module procedure subevt_sort_int
        module procedure subevt_sort_real
```

```
      end interface
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_sort_pdg (subevt, pl)
      type(subevt_t), intent(inout) :: subevt
      type(subevt_t), intent(in) :: pl
      integer :: n
      n = subevt%n_active
      call subevt_sort_int (subevt, pl, abs (3 * subevt%prt(:n)%pdg - 1))
    end subroutine subevt_sort_pdg

    subroutine subevt_sort_int (subevt, pl, ival)
      type(subevt_t), intent(inout) :: subevt
      type(subevt_t), intent(in) :: pl
      integer, dimension(:), intent(in) :: ival
      call subevt_reset (subevt, pl%n_active)
      subevt%n_active = pl%n_active
      subevt%prt = pl%prt( order (ival) )
    end subroutine subevt_sort_int

    subroutine subevt_sort_real (subevt, pl, rval)
      type(subevt_t), intent(inout) :: subevt
      type(subevt_t), intent(in) :: pl
      real(default), dimension(:), intent(in) :: rval
      call subevt_reset (subevt, pl%n_active)
      subevt%n_active = pl%n_active
      subevt%prt = pl%prt( order (rval) )
    end subroutine subevt_sort_real
```

Return the list of particles which have any of the specified PDG codes (and optionally particle type: beam, incoming, outgoing).

The `pack` command was buggy in some gfortran versions, therefore it is unrolled. The unrolled version may be more efficient, actually.

⟨*Subevents: public*⟩+≡
```
    public :: subevt_select_pdg_code
```

⟨*Subevents: procedures*⟩+≡
```
    subroutine subevt_select_pdg_code (subevt, aval, subevt_in, prt_type)
      type(subevt_t), intent(inout) :: subevt
      type(pdg_array_t), intent(in) :: aval
      type(subevt_t), intent(in) :: subevt_in
      integer, intent(in), optional :: prt_type
      integer :: n_active, n_match
      logical, dimension(:), allocatable :: mask
      integer :: i, j
      n_active = subevt_in%n_active
      allocate (mask (n_active))
      forall (i = 1:n_active) &
           mask(i) = aval .match. subevt_in%prt(i)%pdg
      if (present (prt_type)) &
           mask = mask .and. subevt_in%prt(:n_active)%type == prt_type
      n_match = count (mask)
      call subevt_reset (subevt, n_match)
```

```
!!! !!! !!! Workaround for gfortran compiler bug
! subevt%prt(:n_match) = pack (subevt_in%prt(:n_active), mask)
j = 0
do i = 1, n_active
   if (mask(i)) then
      j = j + 1
      subevt%prt(j) = subevt_in%prt(i)
   end if
end do
end subroutine subevt_select_pdg_code
```

5.4.4   Eliminate numerical noise

This is useful for testing purposes: set entries to zero that are smaller in absolute values than a given tolerance parameter.

Note: instead of setting the tolerance in terms of EPSILON (kind-dependent), we fix it to 10^{-16}, which is the typical value for double precision. The reason is that there are situations where intermediate representations (external libraries, files) are limited to double precision, even if the main program uses higher precision.

⟨*Subevents: public*⟩+≡
```
   public :: pacify
```

⟨*Subevents: interfaces*⟩+≡
```
   interface pacify
      module procedure pacify_prt
      module procedure pacify_subevt
   end interface pacify
```

⟨*Subevents: procedures*⟩+≡
```
   subroutine pacify_prt (prt)
     class(prt_t), intent(inout) :: prt
     real(default) :: e
     e = 1e-16_default * energy (prt%p)
     call pacify (prt%p, 10 * e)
     call pacify (prt%p2, 1e4 * e)
   end subroutine pacify_prt

   subroutine pacify_subevt (subevt)
     class(subevt_t), intent(inout) :: subevt
     integer :: i
     do i = 1, subevt%n_active
        call pacify (subevt%prt(i))
     end do
   end subroutine pacify_subevt
```

5.5   User Code Interface

Here we collect interface code that enables the user to inject his own code into the WHIZARD workflow. The code uses data types defined above, and is

referenced in the `expressions` module.

⟨`user_code_interface.f90`⟩≡
  ⟨*File header*⟩

  ```
  module user_code_interface

    use iso_c_binding !NODEP!
  ```
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ```
    use diagnostics !NODEP!
    use c_particles !NODEP!
    use os_interface
  ```

  ⟨*Standard module head*⟩

  ⟨*User Code: public*⟩

  ⟨*User Code: variables*⟩

  ⟨*User Code: interfaces*⟩

  ```
  contains
  ```

  ⟨*User Code: procedures*⟩

  ```
  end module user_code_interface
  ```

5.5.1 User Code Management

This data structure globally holds the filehandle of the user-code library:

⟨*User Code: public*⟩≡
```
    public :: has_user_lib
```

⟨*User Code: variables*⟩≡
```
    type(dlaccess_t), save :: user_lib_handle
    logical, save :: has_user_lib = .false.
    type(string_t), save :: user
```

Compile, link and load user code files. Dlopen all user-provided libraries, included the one just compiled (if any).

By default, we are looking for a library `user.so/dylib`. If this is not present, try `user.f90` and compile it. This can be overridden.

In detail: First, compile all sources explicitly specified on the command line. Then collect all libraries specified on the command line, including `user.so` if it was generated. If there is still no code, check for an existing `user.f90` and compile this. Link everything into a `user.la` libtool library. When done, dlopen all libraries that we have so far.

⟨*User Code: public*⟩+≡
```
    public :: user_code_init
```

403

⟨*User Code: procedures*⟩≡

```
subroutine user_code_init (user_src, user_lib, user_target, rebuild, os_data)
  type(string_t), dimension(:), intent(in) :: user_src, user_lib
  type(string_t), intent(in) :: user_target
  logical, intent(in) :: rebuild
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: user_src_file, user_obj_files, user_lib_file
  logical :: exist
  type(c_funptr) :: fptr
  integer :: i
  call msg_message ("Initializing user code")
  user = user_target;  if (user == "")  user = "user"
  user_obj_files = ""
  inquire (file = char (user) // ".la", exist = exist)
  if (rebuild .or. .not. exist) then
     do i = 1, size (user_src)
        user_src_file = user_src(i) // os_data%fc_src_ext
        inquire (file = char (user_src_file), exist = exist)
        if (exist) then
           call msg_message ("Found user-code source '" &
                // char (user_src_file) // "'.")
           call compile_user_src (user_src_file, user_obj_files)
        else
           call msg_fatal ("User-code source '" // char (user_src_file) &
                // "' not found")
        end if
     end do
     do i = 1, size (user_lib)
        user_lib_file = user_lib(i) // ".la"
        inquire (file = char (user_lib_file), exist = exist)
        if (exist) then
           call msg_message ("Found user-code library '" &
                // char (user_lib_file) // "'.")
        else
           user_lib_file = user_lib(i) // os_data%shlib_ext
           inquire (file = char (user_lib_file), exist = exist)
           if (exist) then
              call msg_message ("Found user-code library '" &
                   // char (user_lib_file) // "'.")
           else
              call msg_fatal ("User-code library '" // char (user_lib(i)) &
                   // "' not found")
           end if
        end if
        user_obj_files = user_obj_files // " " // user_lib_file
     end do
     if (user_obj_files == "") then
        user_src_file = user // os_data%fc_src_ext
        inquire (file = char (user_src_file), exist = exist)
        if (exist) then
           call msg_message ("Found user-code source '" &
                // char (user_src_file) // "'.")
           call compile_user_src (user_src_file, user_obj_files)
        else
```

```
             call msg_fatal ("User-code source '" // char (user_src_file) &
                    // "' not found")
          end if
       end if
       if (user_obj_files /= "") then
          call link_user (char (user), user_obj_files)
       end if
    end if
    call dlaccess_init &
         (user_lib_handle, var_str ("."), &
          user // os_data%shlib_ext, os_data)
    if (dlaccess_has_error (user_lib_handle)) then
       call msg_error (char (dlaccess_get_error (user_lib_handle)))
       call msg_fatal ("Loading user code library '" // char (user) &
            // ".la' failed")
    else
       call msg_message ("User code library '" // char (user) &
            // ".la' successfully loaded")
       has_user_lib = .true.
    end if
  contains
    subroutine compile_user_src (user_src_file, user_obj_files)
      type(string_t), intent(in) :: user_src_file
      type(string_t), intent(inout) :: user_obj_files
      type(string_t) :: basename, ext
      logical :: exist
      basename = user_src_file
      call split (basename, ext, ".", back=.true.)
      if ("." // ext == os_data%fc_src_ext) then
         inquire (file = char (user_src_file), exist = exist)
         if (exist) then
            call msg_message ("Compiling user code file '" &
                 // char (user_src_file) // "'")
            call os_compile_shared (basename, os_data)
            user_obj_files = user_obj_files // " " // basename // ".lo"
         else
            call msg_error ("User code file '" // char (user_src_file) &
                 // "' not found.")
         end if
      else
         call msg_error ("User code file '" // char (user_src_file) &
              // "' should have file extension '" &
              // char (os_data%fc_src_ext) // "'")
      end if
    end subroutine compile_user_src
    subroutine link_user (user_lib, user_obj_files)
      character(*), intent(in) :: user_lib
      type(string_t), intent(in) :: user_obj_files
      call msg_message ("Linking user code library '" &
           // user_lib // char (os_data%shlib_ext) // "'")
      call os_link_shared (user_obj_files, var_str (user_lib), os_data)
    end subroutine link_user
  end subroutine user_code_init
```

Unload all user-code libraries.

⟨*User Code: public*⟩+≡
```
    public :: user_code_final
```

⟨*User Code: procedures*⟩+≡
```
    subroutine user_code_final ()
      if (has_user_lib) then
         call dlaccess_final (user_lib_handle)
         has_user_lib = .false.
      end if
    end subroutine user_code_final
```

Try to load the possible user-defined procedures from the dlopened libraries. If a procedure is not found, do nothing.

⟨*User Code: public*⟩+≡
```
    public :: user_code_find_proc
```

⟨*User Code: procedures*⟩+≡
```
    function user_code_find_proc (name) result (fptr)
      type(string_t), intent(in) :: name
      type(c_funptr) :: fptr
      integer :: i
      fptr = c_null_funptr
      !!! Ticket #529
      ! fptr = libmanager_get_c_funptr (char (user), char (name))
      if (.not. c_associated (fptr)) then
         if (has_user_lib) then
            fptr = dlaccess_get_c_funptr (user_lib_handle, name)
            if (.not. c_associated (fptr)) then
               call msg_fatal ("User procedure '" // char (name) // "' not found")
            end if
         else
            call msg_fatal ("User procedure '" // char (name) &
                 // "' called without user library (missing -u flag?)")
         end if
      end if
    end function user_code_find_proc
```

5.5.2 Interfaces for user-defined functions

The following functions represent user-defined real observables. There may be one or two particles as argument, the result is a real value.

⟨*User Code: public*⟩+≡
```
    public :: user_obs_int_unary
    public :: user_obs_int_binary
    public :: user_obs_real_unary
    public :: user_obs_real_binary
```

⟨*User Code: interfaces*⟩≡
```
    abstract interface
       function user_obs_int_unary (prt1) result (ival) bind(C)
         use iso_c_binding !NODEP!
```

```
        use c_particles !NODEP!
        type(c_prt_t), intent(in) :: prt1
        integer(c_int) :: ival
      end function user_obs_int_unary
    end interface

    abstract interface
      function user_obs_int_binary (prt1, prt2) result (ival) bind(C)
        use iso_c_binding !NODEP!
        use c_particles !NODEP!
        type(c_prt_t), intent(in) :: prt1, prt2
        integer(c_int) :: ival
      end function user_obs_int_binary
    end interface

    abstract interface
      function user_obs_real_unary (prt1) result (rval) bind(C)
        use iso_c_binding !NODEP!
        use c_particles !NODEP!
        type(c_prt_t), intent(in) :: prt1
        real(c_double) :: rval
      end function user_obs_real_unary
    end interface

    abstract interface
      function user_obs_real_binary (prt1, prt2) result (rval) bind(C)
        use iso_c_binding !NODEP!
        use c_particles !NODEP!
        type(c_prt_t), intent(in) :: prt1, prt2
        real(c_double) :: rval
      end function user_obs_real_binary
    end interface
```

The following function takes an array of C-compatible particles and return a single value. The particle array represents a subevent. For C interoperability, we have to use an assumed-size array, hence the array size has to be transferred explicitly.

The cut function returns an int, which we should interpret as a logical value (nonzero=true).

⟨*User Code: public*⟩+≡
```
    public :: user_cut_fun
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
      function user_cut_fun (prt, n_prt) result (iflag) bind(C)
        use iso_c_binding !NODEP!
        use c_particles !NODEP!
        type(c_prt_t), dimension(*), intent(in) :: prt
        integer(c_int), intent(in) :: n_prt
        integer(c_int) :: iflag
      end function user_cut_fun
    end interface
```

The event-shape function returns a real value.

⟨*User Code: public*⟩+≡
```
    public :: user_event_shape_fun
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
       function user_event_shape_fun (prt, n_prt) result (rval) bind(C)
         use iso_c_binding !NODEP!
         use c_particles !NODEP!
         type(c_prt_t), dimension(*), intent(in) :: prt
         integer(c_int), intent(in) :: n_prt
         real(c_double) :: rval
       end function user_event_shape_fun
    end interface
```

5.5.3 Interfaces for user-defined interactions

The following procedure interfaces pertain to user-defined interactions, e.g., spectra or structure functions.

This subroutine retrieves the basic information for setting up the interaction and event generation. All parameters are `intent(inout)`, so we can provide default values. `n_in` and `n_out` are the number of incoming and outgoing particles, respectively. `n_states` is the total number of distinct states that should be generated (counting all states of the incoming particles). `n_col` is the maximal number of color entries a particle can have. `n_dim` is the number of input parameters, i.e., integration dimensions, that the structure function call requires for computing kinematics and matrix elements. `n_var` is the number of variables (e.g., momentum fractions) that the structure function call has to transfer from the kinematics to the dynamics evaluation.

⟨*User Code: public*⟩+≡
```
    public :: user_int_info
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
       subroutine user_int_info (n_in, n_out, n_states, n_col, n_dim, n_var) &
            bind(C)
         use iso_c_binding !NODEP!
         integer(c_int), intent(inout) :: n_in, n_out, n_states, n_col
         integer(c_int), intent(inout) :: n_dim, n_var
       end subroutine user_int_info
    end interface
```

This subroutine retrieves the settings for the quantum number mask of a given particle index in the interaction. A nonzero value indicates that the corresponding quantum number is to be ignored. The lock index is the index of a particle that the current particle is related to. The relation applies if quantum numbers of one of the locked particles are summed over. (This is intended for helicity.)

⟨*User Code: public*⟩+≡
```
    public :: user_int_mask
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
       subroutine user_int_mask (i_prt, m_flv, m_hel, m_col, i_lock) bind(C)
         use iso_c_binding !NODEP!
         integer(c_int), intent(in) :: i_prt
         integer(c_int), intent(inout) :: m_flv, m_hel, m_col, i_lock
       end subroutine user_int_mask
    end interface
```

This subroutine retrieves the quantum numbers for the particle index `i_prt` in state `i_state`. The `flv` value is a PDG code. The `hel` value is an integer helicity (twice the helicity for fermions). The `col` array is an array which has at most `n_col` entries (see above). All parameters are `intent(inout)` since default values exist. In particular, if a mask entry is set by the previous procedure, the corresponding quantum number is ignored anyway.

⟨*User Code: public*⟩+≡
```
    public :: user_int_state
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
       subroutine user_int_state (i_state, i_prt, flv, hel, col) bind(C)
         use iso_c_binding !NODEP!
         integer(c_int), intent(in) :: i_state, i_prt
         integer(c_int), intent(inout) :: flv, hel
         integer(c_int), dimension(*), intent(inout) :: col
       end subroutine user_int_state
    end interface
```

This subroutine takes an array of particle objects with array length `n_in` and an array of input parameters between 0 and 1 with array length `n_dim`. It returns an array of particle objects with array length `n_out`. In addition, it returns an array of internal variables (e.g., momentum fractions, Jacobian) with array length `n_var` that is used by the following subroutine for evaluating the dynamics, i.e., the matrix elements.

⟨*User Code: public*⟩+≡
```
    public :: user_int_kinematics
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
       subroutine user_int_kinematics (prt_in, rval, prt_out, xval) bind(C)
         use iso_c_binding !NODEP!
         use c_particles !NODEP!
         type(c_prt_t), dimension(*), intent(in) :: prt_in
         real(c_double), dimension(*), intent(in) :: rval
         type(c_prt_t), dimension(*), intent(inout) :: prt_out
         real(c_double), dimension(*), intent(out) :: xval
       end subroutine user_int_kinematics
    end interface
```

This subroutine takes the array of variables (e.g., momentum fractions) with length `n_var` which has been generated by the previous subroutine and a real variable, the energy scale of the event. It returns an array of matrix-element

values, one entry for each quantum state `n_states`. The ordering of matrix elements must correspond to the ordering of states.

⟨*User Code: public*⟩+≡
```
    public :: user_int_evaluate
```

⟨*User Code: interfaces*⟩+≡
```
    abstract interface
       subroutine user_int_evaluate (xval, scale, fval) bind(C)
         use iso_c_binding !NODEP!
         real(c_double), dimension(*), intent(in) :: xval
         real(c_double), intent(in) :: scale
         real(c_double), dimension(*), intent(out) :: fval
       end subroutine user_int_evaluate
    end interface
```

5.6 Variables

The user interface deals with variables that are handled similar to full-flegded programming languages. The system will add a lot of predefined variables (model parameters, flags, etc.) that are accessible to the user by the same methods.

Variables can be of various type: logical (boolean/flag), integer, real (default precision), subevents (used in cut expressions), arrays of PDG codes (aliases for particles), strings. Furthermore, in cut expressions we have unary and binary observables, which are used like real parameters but behave like functions.

⟨`variables.f90`⟩≡
  ⟨*File header*⟩

  `module variables`

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use limits, only: FMT_14, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use pdg_arrays
    use subevents
```

  ⟨*Standard module head*⟩

  ⟨*Variables: public*⟩

  ⟨*Variables: parameters*⟩

  ⟨*Variables: types*⟩

  ⟨*Variables: interfaces*⟩

  `contains`

⟨*Variables: procedures*⟩

```
end module variables
```

5.6.1 Variable list entries

Variable (and constant) values can be of one of the following types:

⟨*Variables: parameters*⟩≡
```
    integer, parameter, public :: V_NONE = 0, V_LOG = 1, V_INT = 2, V_REAL = 3
    integer, parameter, public :: V_CMPLX = 4, V_SEV = 5, V_PDG = 6, V_STR = 7
    integer, parameter, public :: V_OBS1_INT = 11, V_OBS2_INT = 12
    integer, parameter, public :: V_OBS1_REAL = 21, V_OBS2_REAL = 22
    integer, parameter, public :: V_UOBS1_INT = 31, V_UOBS2_INT = 32
    integer, parameter, public :: V_UOBS1_REAL = 41, V_UOBS2_REAL = 42
```

The type

This is an entry in the variable list. It can be of any type; in each case only one value is allocated. It may be physically allocated upon creation, in which case `is_allocated` is true, or it may contain just a pointer to a value somewhere else, in which case `is_allocated` is false.

The flag `is_defined` is set when the variable is given a value, even the undefined value. (Therefore it is distinct from `is_known`.) This matters for variable declaration in the SINDARIN language. The variable is set up in the compilation step and initially marked as defined, but after compilation all variables are set undefined. Each variable becomes defined when it is explicitly set. The difference matters in loops.

`is_locked` means that it cannot be given a value using the interface routines `var_list_set_XXX` below. It can only be initialized, or change automatically due to a side effect.

`is_copy` means that this is a local copy of a global variable. The copy has a pointer to the original, which can be used to restore a previous value.

`is_intrinsic` means that this variable is defined by the program, not by the user. Intrinsic variables cannot be (re)declared, but their values can be reset unless they are locked. `is_user_var` means that the variable has been declared by the user. It could be a new variable, or a local copy of an intrinsic variable.

The flag `is_known` is a pointer which parallels the use of the value pointer. For pointer variables, it is set if the value should point to a known value. For ordinary variables, it should be true.

The value is implemented as a set of alternative type-specific pointers. This emulates polymorphism, and it allows for actual pointer variables. Observable-type variables have function pointers as values, so they behave like macros. The functions make use of the particle objects accessible via the pointers `prt1` and `prt2`.

Finally, the `next` pointer indicates that we are making lists of variables. A more efficient implementation might switch to hashes or similar; the current implementation has $O(N)$ lookup.

⟨*Variables: public*⟩≡
```
    public :: var_entry_t
```

⟨Variables: types⟩≡

```
type :: var_entry_t
   private
   integer :: type = V_NONE
   type(string_t) :: name
   logical :: is_allocated = .false.
   logical :: is_defined = .false.
   logical :: is_locked = .false.
   logical :: is_copy = .false.
   type(var_entry_t), pointer :: original => null ()
   logical :: is_intrinsic = .false.
   logical :: is_user_var = .false.
   logical, pointer :: is_known => null ()
   logical,          pointer :: lval => null ()
   integer,          pointer :: ival => null ()
   real(default),    pointer :: rval => null ()
   complex(default), pointer :: cval => null ()
   type(subevt_t),  pointer :: pval => null ()
   type(pdg_array_t), pointer :: aval => null ()
   type(string_t),   pointer :: sval => null ()
   procedure(obs_unary_int),   nopass, pointer :: obs1_int  => null ()
   procedure(obs_unary_real),  nopass, pointer :: obs1_real => null ()
   procedure(obs_binary_int),  nopass, pointer :: obs2_int  => null ()
   procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
   type(prt_t), pointer :: prt1 => null ()
   type(prt_t), pointer :: prt2 => null ()
   type(var_entry_t), pointer :: next => null ()
end type var_entry_t
```

Interfaces for the observable functions

⟨Variables: public⟩+≡

```
public :: obs_unary_int
public :: obs_unary_real
public :: obs_binary_int
public :: obs_binary_real
```

⟨Variables: interfaces⟩≡

```
abstract interface
   function obs_unary_int (prt1) result (ival)
     import
     integer :: ival
     type(prt_t), intent(in) :: prt1
   end function obs_unary_int
end interface
abstract interface
   function obs_unary_real (prt1) result (rval)
     import
     real(default) :: rval
     type(prt_t), intent(in) :: prt1
   end function obs_unary_real
end interface
abstract interface
```

```
        function obs_binary_int (prt1, prt2) result (ival)
          import
          integer :: ival
          type(prt_t), intent(in) :: prt1, prt2
        end function obs_binary_int
    end interface
    abstract interface
        function obs_binary_real (prt1, prt2) result (rval)
          import
          real(default) :: rval
          type(prt_t), intent(in) :: prt1, prt2
        end function obs_binary_real
    end interface
```

Initialization

Initialize an entry, optionally with a physical value. We also allocate the
is_known flag and set it if the value is set.

⟨*Variables: public*⟩+≡
```
    public :: var_entry_init_log
    public :: var_entry_init_int
    public :: var_entry_init_real
    public :: var_entry_init_cmplx
    public :: var_entry_init_pdg_array
    public :: var_entry_init_subevt
    public :: var_entry_init_string
```

⟨*Variables: procedures*⟩≡
```
    subroutine var_entry_init_log (var, name, lval, intrinsic, user)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      logical, intent(in), optional :: lval
      logical, intent(in), optional :: intrinsic, user
      var%name = name
      var%type = V_LOG
      allocate (var%lval, var%is_known)
      if (present (lval)) then
         var%lval = lval
         var%is_defined = .true.
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      if (present (user))  var%is_user_var = user
      var%is_allocated = .true.
    end subroutine var_entry_init_log

    subroutine var_entry_init_int (var, name, ival, intrinsic, user)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: ival
      logical, intent(in), optional :: intrinsic, user
```

```
      var%name = name
      var%type = V_INT
      allocate (var%ival, var%is_known)
      if (present (ival)) then
         var%ival = ival
         var%is_defined = .true.
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      if (present (user))  var%is_user_var = user
      var%is_allocated = .true.
    end subroutine var_entry_init_int

    subroutine var_entry_init_real (var, name, rval, intrinsic, user)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      real(default), intent(in), optional :: rval
      logical, intent(in), optional :: intrinsic, user
      var%name = name
      var%type = V_REAL
      allocate (var%rval, var%is_known)
      if (present (rval)) then
         var%rval = rval
         var%is_defined = .true.
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      if (present (user))  var%is_user_var = user
      var%is_allocated = .true.
    end subroutine var_entry_init_real

    subroutine var_entry_init_cmplx (var, name, cval, intrinsic, user)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      complex(default), intent(in), optional :: cval
      logical, intent(in), optional :: intrinsic, user
      var%name = name
      var%type = V_CMPLX
      allocate (var%cval, var%is_known)
      if (present (cval)) then
         var%cval = cval
         var%is_defined = .true.
         var%is_known = .true.
      else
         var%is_known = .false.
      end if
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      if (present (user))  var%is_user_var = user
      var%is_allocated = .true.
    end subroutine var_entry_init_cmplx
```

```fortran
subroutine var_entry_init_subevt (var, name, pval, intrinsic, user)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(subevt_t), intent(in), optional :: pval
  logical, intent(in), optional :: intrinsic, user
  var%name = name
  var%type = V_SEV
  allocate (var%pval, var%is_known)
  if (present (pval)) then
     var%pval = pval
     var%is_defined = .true.
     var%is_known = .true.
  else
     var%is_known = .false.
  end if
  if (present (intrinsic))  var%is_intrinsic = intrinsic
  if (present (user))  var%is_user_var = user
  var%is_allocated = .true.
end subroutine var_entry_init_subevt

subroutine var_entry_init_pdg_array (var, name, aval, intrinsic, user)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(pdg_array_t), intent(in), optional :: aval
  logical, intent(in), optional :: intrinsic, user
  var%name = name
  var%type = V_PDG
  allocate (var%aval, var%is_known)
  if (present (aval)) then
     var%aval = aval
     var%is_defined = .true.
     var%is_known = .true.
  else
     var%is_known = .false.
  end if
  if (present (intrinsic))  var%is_intrinsic = intrinsic
  if (present (user))  var%is_user_var = user
  var%is_allocated = .true.
end subroutine var_entry_init_pdg_array

subroutine var_entry_init_string (var, name, sval, intrinsic, user)
  type(var_entry_t), intent(out) :: var
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: sval
  logical, intent(in), optional :: intrinsic, user
  var%name = name
  var%type = V_STR
  allocate (var%sval, var%is_known)
  if (present (sval)) then
     var%sval = sval
     var%is_defined = .true.
     var%is_known = .true.
  else
```

```
          var%is_known = .false.
       end if
       if (present (intrinsic))  var%is_intrinsic = intrinsic
       if (present (user))  var%is_user_var = user
       var%is_allocated = .true.
    end subroutine var_entry_init_string
```

Initialize an entry with a pointer to the value and, for numeric/logical values, a
pointer to the `is_known` flag.

⟨*Variables: public*⟩+≡
```
    public :: var_entry_init_log_ptr
    public :: var_entry_init_int_ptr
    public :: var_entry_init_real_ptr
    public :: var_entry_init_cmplx_ptr
    public :: var_entry_init_pdg_array_ptr
    public :: var_entry_init_subevt_ptr
    public :: var_entry_init_string_ptr
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_init_log_ptr (var, name, lval, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      logical, intent(in), target :: lval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_LOG
      var%lval => lval
      var%is_known => is_known
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      var%is_defined = .true.
    end subroutine var_entry_init_log_ptr

    subroutine var_entry_init_int_ptr (var, name, ival, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      integer, intent(in), target :: ival
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
      var%type = V_INT
      var%ival => ival
      var%is_known => is_known
      if (present (intrinsic))  var%is_intrinsic = intrinsic
      var%is_defined = .true.
    end subroutine var_entry_init_int_ptr

    subroutine var_entry_init_real_ptr (var, name, rval, is_known, intrinsic)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      real(default), intent(in), target :: rval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: intrinsic
      var%name = name
```

```
    var%type = V_REAL
    var%rval => rval
    var%is_known => is_known
    if (present (intrinsic))  var%is_intrinsic = intrinsic
    var%is_defined = .true.
  end subroutine var_entry_init_real_ptr

  subroutine var_entry_init_cmplx_ptr (var, name, cval, is_known, intrinsic)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    complex(default), intent(in), target :: cval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_CMPLX
    var%cval => cval
    var%is_known => is_known
    if (present (intrinsic))  var%is_intrinsic = intrinsic
    var%is_defined = .true.
  end subroutine var_entry_init_cmplx_ptr

  subroutine var_entry_init_pdg_array_ptr (var, name, aval, is_known, intrinsic)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_PDG
    var%aval => aval
    var%is_known => is_known
    if (present (intrinsic))  var%is_intrinsic = intrinsic
    var%is_defined = .true.
  end subroutine var_entry_init_pdg_array_ptr

  subroutine var_entry_init_subevt_ptr (var, name, pval, is_known, intrinsic)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), target :: pval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_SEV
    var%pval => pval
    var%is_known => is_known
    if (present (intrinsic))  var%is_intrinsic = intrinsic
    var%is_defined = .true.
  end subroutine var_entry_init_subevt_ptr

  subroutine var_entry_init_string_ptr (var, name, sval, is_known, intrinsic)
    type(var_entry_t), intent(out) :: var
    type(string_t), intent(in) :: name
    type(string_t), intent(in), target :: sval
    logical, intent(in), target :: is_known
```

```
    logical, intent(in), optional :: intrinsic
    var%name = name
    var%type = V_STR
    var%sval => sval
    var%is_known => is_known
    if (present (intrinsic))  var%is_intrinsic = intrinsic
    var%is_defined = .true.
  end subroutine var_entry_init_string_ptr
```

Initialize an entry with an observable. The procedure pointer is not yet set.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_init_obs (var, name, type, prt1, prt2)
      type(var_entry_t), intent(out) :: var
      type(string_t), intent(in) :: name
      integer, intent(in) :: type
      type(prt_t), intent(in), target :: prt1
      type(prt_t), intent(in), optional, target :: prt2
      var%type = type
      var%name = name
      var%prt1 => prt1
      if (present (prt2))  var%prt2 => prt2
      var%is_intrinsic = .true.
      var%is_defined = .true.
    end subroutine var_entry_init_obs
```

Mark an entry as undefined it it is a user-defined variable object, so force re-initialization.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_undefine (var)
      type(var_entry_t), intent(inout) :: var
      var%is_defined = .not. var%is_user_var
      var%is_known = var%is_defined .and. var%is_known
    end subroutine var_entry_undefine
```

Clear an entry: mark it as unknown.

⟨*Variables: public*⟩+≡
```
    public :: var_entry_clear
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_clear (var)
      type(var_entry_t), intent(inout) :: var
      var%is_known = .false.
    end subroutine var_entry_clear
```

Lock an entry: forbid resetting the entry after initialization.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_lock (var, locked)
      type(var_entry_t), intent(inout) :: var
      logical, intent(in), optional :: locked
      if (present (locked)) then
         var%is_locked = locked
      else
```

```
      var%is_locked = .true.
    end if
  end subroutine var_entry_lock
```

Finalizer

⟨*Variables: public*⟩+≡
```
    public :: var_entry_final
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_final (var)
      type(var_entry_t), intent(inout) :: var
      if (var%is_allocated) then
         select case (var%type)
         case (V_LOG); deallocate (var%lval)
         case (V_INT); deallocate (var%ival)
         case (V_REAL);deallocate (var%rval)
         case (V_CMPLX);deallocate (var%cval)
         case (V_SEV); deallocate (var%pval)
         case (V_PDG); deallocate (var%aval)
         case (V_STR); deallocate (var%sval)
         end select
         deallocate (var%is_known)
         var%is_allocated = .false.
         var%is_defined = .false.
      end if
    end subroutine var_entry_final
```

Output

⟨*Variables: public*⟩+≡
```
    public :: var_entry_write
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_entry_write (var, unit, model_name, show_ptr, &
        intrinsic, pacified)
      type(var_entry_t), intent(in) :: var
      integer, intent(in), optional :: unit
      type(string_t), intent(in), optional :: model_name
      logical, intent(in), optional :: show_ptr
      logical, intent(in), optional :: intrinsic
      logical, intent(in), optional :: pacified
      logical :: num_pac
      real(default) :: rval
      complex(default) :: cval
      integer :: u
      character(len=7) :: fmt
      call pac_fmt (fmt, FMT_19, FMT_14, pacified)
      u = output_unit (unit);  if (u < 0)  return
      if (present (intrinsic)) then
         if (var%is_intrinsic .neqv. intrinsic)  return
      end if
      if (.not. var%is_defined) then
```

```
      write (u, "(A,1x)", advance="no")  "[undefined]"
end if
if (.not. var%is_intrinsic) then
   write (u, "(A,1x)", advance="no")  "[user variable]"
end if
num_pac = .false.
if (present (pacified))  num_pac = pacified
if (associated (var%original)) then
   if (present (model_name)) then
      write (u, "(A,A)", advance="no")  char(model_name), "."
   end if
end if
write (u, "(A)", advance="no")  char (var%name)
if (var%is_locked)  write (u, "(A)", advance="no")  "*"
if (var%is_allocated) then
   write (u, "(A)", advance="no")  " = "
else if (var%type /= V_NONE) then
   write (u, "(A)", advance="no")  " => "
end if
select case (var%type)
case (V_NONE); write (u, *)
case (V_LOG)
   if (var%is_known) then
      if (var%lval) then
         write (u, "(A)")  "true"
      else
         write (u, "(A)")  "false"
      end if
   else
      write (u, "(A)")  "[unknown logical]"
   end if
case (V_INT)
   if (var%is_known) then
      write (u, "(I0)")  var%ival
   else
      write (u, "(A)")  "[unknown integer]"
   end if
case (V_REAL)
   if (var%is_known) then
      rval = var%rval
      if (num_pac) then
         call pacify (rval, 10 * epsilon (1._default))
      end if
      write (u, "(" // fmt // ")")  rval
   else
      write (u, "(A)")  "[unknown real]"
   end if
case (V_CMPLX)
   if (var%is_known) then
      cval = var%cval
      if (num_pac) then
         call pacify (cval, 10 * epsilon (1._default))
      end if
      write (u, "('('," // fmt // ",',',"  // fmt // ",')')")  cval
```

```
            else
               write (u, "(A)")  "[unknown complex]"
            end if
         case (V_SEV)
            if (var%is_known) then
               call subevt_write (var%pval, unit, prefix="         ")
            else
               write (u, "(A)")  "[unknown subevent]"
            end if
         case (V_PDG)
            if (var%is_known) then
               call pdg_array_write (var%aval, u);  write (u, *)
            else
               write (u, "(A)")  "[unknown PDG array]"
            end if
         case (V_STR)
            if (var%is_known) then
               write (u, "(A)")  '"' // char (var%sval) // '"'
            else
               write (u, "(A)")  "[unknown string]"
            end if
         case (V_OBS1_INT);  write (u, *) "[int] = unary observable"
         case (V_OBS2_INT);  write (u, *) "[int] = binary observable"
         case (V_OBS1_REAL); write (u, *) "[real] = unary observable"
         case (V_OBS2_REAL); write (u, *) "[real] = binary observable"
         case (V_UOBS1_INT);  write (u, *) "[int] = unary user observable"
         case (V_UOBS2_INT);  write (u, *) "[int] = binary user observable"
         case (V_UOBS1_REAL); write (u, *) "[real] = unary user observable"
         case (V_UOBS2_REAL); write (u, *) "[real] = binary user observable"
         end select
         if (present (show_ptr)) then
            if (show_ptr .and. var%is_copy .and. associated (var%original)) then
               write (u, "('  => ')", advance="no")
               call var_entry_write (var%original, unit)
            end if
         end if
      end subroutine var_entry_write
```

Accessing contents

⟨*Variables: public*⟩+≡
```
    public :: var_entry_get_name
    public :: var_entry_get_type
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_get_name (var) result (name)
      type(string_t) :: name
      type(var_entry_t), intent(in) :: var
      name = var%name
    end function var_entry_get_name

    function var_entry_get_type (var) result (type)
      integer :: type
```

```
    type(var_entry_t), intent(in) :: var
    type = var%type
end function var_entry_get_type
```

Return true if the variable is defined. This the case if it is allocated and known, or if it is a pointer.

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_defined
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_defined (var) result (defined)
      logical :: defined
      type(var_entry_t), intent(in) :: var
      defined = var%is_defined
    end function var_entry_is_defined
```

Return true if the variable is locked

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_locked
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_locked (var) result (locked)
      logical :: locked
      type(var_entry_t), intent(in) :: var
      locked = var%is_locked
    end function var_entry_is_locked
```

Return true if the variable is intrinsic

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_intrinsic
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_intrinsic (var) result (flag)
      logical :: flag
      type(var_entry_t), intent(in) :: var
      flag = var%is_intrinsic
    end function var_entry_is_intrinsic
```

Return true if the variable is a copy

⟨*Variables: public*⟩+≡
```
    public :: var_entry_is_copy
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_is_copy (var) result (flag)
      logical :: flag
      type(var_entry_t), intent(in) :: var
      flag = var%is_copy
    end function var_entry_is_copy
```

Return components

⟨Variables: public⟩+≡

```
    public :: var_entry_is_known
    public :: var_entry_get_lval
    public :: var_entry_get_ival
    public :: var_entry_get_rval
    public :: var_entry_get_cval
    public :: var_entry_get_aval
    public :: var_entry_get_pval
    public :: var_entry_get_sval
```

⟨Variables: procedures⟩+≡

```
    function var_entry_is_known (var) result (flag)
      logical :: flag
      type(var_entry_t), intent(in) :: var
      flag = var%is_known
    end function var_entry_is_known

    function var_entry_get_lval (var) result (lval)
      logical :: lval
      type(var_entry_t), intent(in) :: var
      lval = var%lval
    end function var_entry_get_lval

    function var_entry_get_ival (var) result (ival)
      integer :: ival
      type(var_entry_t), intent(in) :: var
      ival = var%ival
    end function var_entry_get_ival

    function var_entry_get_rval (var) result (rval)
      real(default) :: rval
      type(var_entry_t), intent(in) :: var
      rval = var%rval
    end function var_entry_get_rval

    function var_entry_get_cval (var) result (cval)
      complex(default) :: cval
      type(var_entry_t), intent(in) :: var
      cval = var%cval
    end function var_entry_get_cval

    function var_entry_get_aval (var) result (aval)
      type(pdg_array_t) :: aval
      type(var_entry_t), intent(in) :: var
      aval = var%aval
    end function var_entry_get_aval

    function var_entry_get_pval (var) result (pval)
      type(subevt_t) :: pval
      type(var_entry_t), intent(in) :: var
      pval = var%pval
    end function var_entry_get_pval

    function var_entry_get_sval (var) result (sval)
```

```
    type(string_t) :: sval
    type(var_entry_t), intent(in) :: var
    sval = var%sval
  end function var_entry_get_sval
```

Return pointers to components

⟨*Variables: public*⟩+≡
```
    public :: var_entry_get_known_ptr
    public :: var_entry_get_lval_ptr
    public :: var_entry_get_ival_ptr
    public :: var_entry_get_rval_ptr
    public :: var_entry_get_cval_ptr
    public :: var_entry_get_aval_ptr
    public :: var_entry_get_pval_ptr
    public :: var_entry_get_sval_ptr
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_get_known_ptr (var) result (ptr)
      logical, pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%is_known
    end function var_entry_get_known_ptr

    function var_entry_get_lval_ptr (var) result (ptr)
      logical, pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%lval
    end function var_entry_get_lval_ptr

    function var_entry_get_ival_ptr (var) result (ptr)
      integer, pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%ival
    end function var_entry_get_ival_ptr

    function var_entry_get_rval_ptr (var) result (ptr)
      real(default), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%rval
    end function var_entry_get_rval_ptr

    function var_entry_get_cval_ptr (var) result (ptr)
      complex(default), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%cval
    end function var_entry_get_cval_ptr

    function var_entry_get_pval_ptr (var) result (ptr)
      type(subevt_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%pval
    end function var_entry_get_pval_ptr

    function var_entry_get_aval_ptr (var) result (ptr)
```

```
      type(pdg_array_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%aval
    end function var_entry_get_aval_ptr

    function var_entry_get_sval_ptr (var) result (ptr)
      type(string_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%sval
    end function var_entry_get_sval_ptr
```

⟨*Variables: public*⟩+≡
```
    public :: var_entry_get_prt1_ptr
    public :: var_entry_get_prt2_ptr
```

⟨*Variables: procedures*⟩+≡
```
    function var_entry_get_prt1_ptr (var) result (ptr)
      type(prt_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%prt1
    end function var_entry_get_prt1_ptr

    function var_entry_get_prt2_ptr (var) result (ptr)
      type(prt_t), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%prt2
    end function var_entry_get_prt2_ptr
```

We would like to also use functions here (for consistency), but a nagfor bug
temporarily forces use to use subroutines.

⟨*Variables: public*⟩+≡
```
    public :: var_entry_assign_obs1_int_ptr
    public :: var_entry_assign_obs1_real_ptr
    public :: var_entry_assign_obs2_int_ptr
    public :: var_entry_assign_obs2_real_ptr
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_assign_obs1_int_ptr (ptr, var)
      procedure(obs_unary_int), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%obs1_int
    end subroutine var_entry_assign_obs1_int_ptr

    subroutine var_entry_assign_obs1_real_ptr (ptr, var)
      procedure(obs_unary_real), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%obs1_real
    end subroutine var_entry_assign_obs1_real_ptr

    subroutine var_entry_assign_obs2_int_ptr (ptr, var)
      procedure(obs_binary_int), pointer :: ptr
      type(var_entry_t), intent(in), target :: var
      ptr => var%obs2_int
    end subroutine var_entry_assign_obs2_int_ptr
```

```
subroutine var_entry_assign_obs2_real_ptr (ptr, var)
  procedure(obs_binary_real), pointer :: ptr
  type(var_entry_t), intent(in), target :: var
  ptr => var%obs2_real
end subroutine var_entry_assign_obs2_real_ptr
```

5.6.2   Setting values

Undefine the value.

⟨Variables: procedures⟩+≡
```
subroutine var_entry_clear_value (var)
  type(var_entry_t), intent(inout) :: var
  var%is_known = .false.
end subroutine var_entry_clear_value
```

⟨Variables: public⟩+≡
```
public :: var_entry_set_log
public :: var_entry_set_int
public :: var_entry_set_real
public :: var_entry_set_cmplx
public :: var_entry_set_pdg_array
public :: var_entry_set_subevt
public :: var_entry_set_string
```
⟨Variables: procedures⟩+≡
```
recursive subroutine var_entry_set_log &
     (var, lval, is_known, verbose, model_name)
  type(var_entry_t), intent(inout) :: var
  logical, intent(in) :: lval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%lval = lval
  var%is_known = is_known
  var%is_defined = .true.
  if (associated (var%original)) then
     call var_entry_set_log (var%original, lval, is_known)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        if (u >= 0) flush (u)
     end if
  end if
end subroutine var_entry_set_log

recursive subroutine var_entry_set_int &
     (var, ival, is_known, verbose, model_name)
```

```
  type(var_entry_t), intent(inout) :: var
  integer, intent(in) :: ival
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%ival = ival
  var%is_known = is_known
  var%is_defined = .true.
  if (associated (var%original)) then
     call var_entry_set_int (var%original, ival, is_known)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write (var, model_name=model_name)
        call var_entry_write (var, model_name=model_name, unit=u)
        if (u >= 0) flush (u)
     end if
  end if
end subroutine var_entry_set_int

recursive subroutine var_entry_set_real &
     (var, rval, is_known, verbose, model_name, pacified)
  type(var_entry_t), intent(inout) :: var
  real(default), intent(in) :: rval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose, pacified
  type(string_t), intent(in), optional :: model_name
  integer :: u
  u = logfile_unit ()
  var%rval = rval
  var%is_known = is_known
  var%is_defined = .true.
  if (associated (var%original)) then
     call var_entry_set_real &
          (var%original, rval, is_known, pacified = pacified)
  end if
  if (present (verbose)) then
     if (verbose) then
        call var_entry_write &
             (var, model_name=model_name, pacified = pacified)
        call var_entry_write &
             (var, model_name=model_name, unit=u, pacified = pacified)
        if (u >= 0) flush (u)
     end if
  end if
end subroutine var_entry_set_real

recursive subroutine var_entry_set_cmplx &
     (var, cval, is_known, verbose, model_name, pacified)
  type(var_entry_t), intent(inout) :: var
  complex(default), intent(in) :: cval
  logical, intent(in) :: is_known
```

```
    logical, intent(in), optional :: verbose, pacified
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%cval = cval
    var%is_known = is_known
    var%is_defined = .true.
    if (associated (var%original)) then
       call var_entry_set_cmplx &
            (var%original, cval, is_known, pacified = pacified)
    end if
    if (present (verbose)) then
       if (verbose) then
          call var_entry_write &
               (var, model_name=model_name, pacified = pacified)
          call var_entry_write &
               (var, model_name=model_name, unit=u, pacified = pacified)
          if (u >= 0) flush (u)
       end if
    end if
  end subroutine var_entry_set_cmplx

  recursive subroutine var_entry_set_pdg_array &
       (var, aval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    type(pdg_array_t), intent(in) :: aval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
    u = logfile_unit ()
    var%aval = aval
    var%is_known = is_known
    var%is_defined = .true.
    if (associated (var%original)) then
       call var_entry_set_pdg_array (var%original, aval, is_known)
    end if
    if (present (verbose)) then
       if (verbose) then
          call var_entry_write (var, model_name=model_name)
          call var_entry_write (var, model_name=model_name, unit=u)
          if (u >= 0) flush (u)
       end if
    end if
  end subroutine var_entry_set_pdg_array

  recursive subroutine var_entry_set_subevt &
       (var, pval, is_known, verbose, model_name)
    type(var_entry_t), intent(inout) :: var
    type(subevt_t), intent(in) :: pval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    type(string_t), intent(in), optional :: model_name
    integer :: u
```

```
      u = logfile_unit ()
      var%pval = pval
      var%is_known = is_known
      var%is_defined = .true.
      if (associated (var%original)) then
         call var_entry_set_subevt (var%original, pval, is_known)
      end if
      if (present (verbose)) then
         if (verbose) then
            call var_entry_write (var, model_name=model_name)
            call var_entry_write (var, model_name=model_name, unit=u)
            if (u >= 0) flush (u)
         end if
      end if
    end subroutine var_entry_set_subevt

    recursive subroutine var_entry_set_string &
         (var, sval, is_known, verbose, model_name)
      type(var_entry_t), intent(inout) :: var
      type(string_t), intent(in) :: sval
      logical, intent(in) :: is_known
      logical, intent(in), optional :: verbose
      type(string_t), intent(in), optional :: model_name
      integer :: u
      u = logfile_unit ()
      var%sval = sval
      var%is_known = is_known
      var%is_defined = .true.
      if (associated (var%original)) then
         call var_entry_set_string (var%original, sval, is_known)
      end if
      if (present (verbose)) then
         if (verbose) then
            call var_entry_write (var, model_name=model_name)
            call var_entry_write (var, model_name=model_name, unit=u)
            if (u >= 0) flush (u)
         end if
      end if
    end subroutine var_entry_set_string
```

5.6.3   Copies and pointer variables

Initialize an entry with a copy of an existing variable entry. The copy is physically allocated with the same type as the original.

⟨*Variables: procedures*⟩+≡

```
    subroutine var_entry_init_copy (var, original, user)
      type(var_entry_t), intent(out) :: var
      type(var_entry_t), intent(in), target :: original
      logical, intent(in), optional :: user
      type(string_t) :: name
      logical :: intrinsic
      name = var_entry_get_name (original)
```

```
      intrinsic = original%is_intrinsic
      select case (original%type)
      case (V_LOG)
         call var_entry_init_log (var, name, intrinsic=intrinsic, user=user)
      case (V_INT)
         call var_entry_init_int (var, name, intrinsic=intrinsic, user=user)
      case (V_REAL)
         call var_entry_init_real (var, name, intrinsic=intrinsic, user=user)
      case (V_CMPLX)
         call var_entry_init_cmplx (var, name, intrinsic=intrinsic, user=user)
      case (V_SEV)
         call var_entry_init_subevt (var, name, intrinsic=intrinsic, user=user)
      case (V_PDG)
         call var_entry_init_pdg_array (var, name, intrinsic=intrinsic, user=user)
      case (V_STR)
         call var_entry_init_string (var, name, intrinsic=intrinsic, user=user)
      end select
      var%is_copy = .true.
    end subroutine var_entry_init_copy
```

Clear the pointer to the original.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_clear_original_pointer (var)
      type(var_entry_t), intent(inout) :: var
      var%original => null ()
    end subroutine var_entry_clear_original_pointer
```

Set the pointer to the original. For a free parameter, the variable holds both the value and the pointer. For a derived parameter, we associate the pointer directly. Derived parameters thus need not be synchronized explicitly.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_set_original_pointer (var, original)
      type(var_entry_t), intent(inout) :: var
      type(var_entry_t), intent(in), target :: original
      if (var_entry_is_locked (original)) then
         call var_entry_lock (var)
      end if
      var%original => original
    end subroutine var_entry_set_original_pointer
```

Synchronize a variable with its original: if a pointer exists, set the value to be equal to value pointed to.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_entry_synchronize (var)
      type(var_entry_t), intent(inout) :: var
      if (associated (var%original)) then
         var%is_defined = var%original%is_defined
         var%is_known = var%original%is_known
         if (var%original%is_known) then
            select case (var%type)
            case (V_LOG);  var%lval = var%original%lval
            case (V_INT);  var%ival = var%original%ival
```

```
          case (V_REAL); var%rval = var%original%rval
          case (V_CMPLX); var%cval = var%original%cval
          case (V_SEV);  var%pval = var%original%pval
          case (V_PDG);  var%aval = var%original%aval
          case (V_STR);  var%sval = var%original%sval
          end select
       end if
    end if
  end subroutine var_entry_synchronize
```

Restore the previous value of the original, using the value stored in the variable. This is a side-effect operation.

⟨*Variables: procedures*⟩+≡
```
  subroutine var_entry_restore (var)
    type(var_entry_t), intent(inout) :: var
    if (associated (var%original)) then
       if (var%is_known) then
          select case (var%type)
          case (V_LOG);  var%original%lval = var%lval
          case (V_INT);  var%original%ival = var%ival
          case (V_REAL); var%original%rval = var%rval
          case (V_CMPLX); var%original%cval = var%cval
          case (V_SEV);  var%original%pval = var%pval
          case (V_PDG);  var%original%aval = var%aval
          case (V_STR);  var%original%sval = var%sval
          end select
       end if
    end if
  end subroutine var_entry_restore
```

5.6.4 Variable lists

The type

Variable lists can be linked together. No initializer needed. They are deleted separately.

⟨*Variables: public*⟩+≡
```
  public :: var_list_t
```

⟨*Variables: types*⟩+≡
```
  type :: var_list_t
    private
    type(var_entry_t), pointer :: first => null ()
    type(var_entry_t), pointer :: last => null ()
    type(var_list_t), pointer :: next => null ()
  end type var_list_t
```

Constructors

⟨*Variables: public*⟩+≡
```
  public :: var_list_link
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_link (var_list, next)
      type(var_list_t), intent(inout) :: var_list
      type(var_list_t), intent(in), target :: next
      var_list%next => next
    end subroutine var_list_link
```

Append a new entry to an existing list.

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_append (var_list, var, verbose)
      type(var_list_t), intent(inout) :: var_list
      type(var_entry_t), intent(in), target :: var
      logical, intent(in), optional :: verbose
      if (associated (var_list%last)) then
         var_list%last%next => var
      else
         var_list%first => var
      end if
      var_list%last => var
      if (present (verbose)) then
         if (verbose)  call var_entry_write (var)
      end if
    end subroutine var_list_append
```

⟨*Variables: public*⟩+≡
```
    public :: var_list_append_log
    public :: var_list_append_int
    public :: var_list_append_real
    public :: var_list_append_cmplx
    public :: var_list_append_subevt
    public :: var_list_append_pdg_array
    public :: var_list_append_string
```
⟨*Variables: interfaces*⟩+≡
```
    interface var_list_append_log
       module procedure var_list_append_log_s
       module procedure var_list_append_log_c
    end interface
    interface var_list_append_int
       module procedure var_list_append_int_s
       module procedure var_list_append_int_c
    end interface
    interface var_list_append_real
       module procedure var_list_append_real_s
       module procedure var_list_append_real_c
    end interface
    interface var_list_append_cmplx
       module procedure var_list_append_cmplx_s
       module procedure var_list_append_cmplx_c
    end interface
    interface var_list_append_subevt
       module procedure var_list_append_subevt_s
       module procedure var_list_append_subevt_c
    end interface
```

```
      interface var_list_append_pdg_array
         module procedure var_list_append_pdg_array_s
         module procedure var_list_append_pdg_array_c
      end interface
      interface var_list_append_string
         module procedure var_list_append_string_s
         module procedure var_list_append_string_c
      end interface
```
⟨*Variables: procedures*⟩+≡
```
      subroutine var_list_append_log_s &
           (var_list, name, lval, locked, verbose, intrinsic, user)
        type(var_list_t), intent(inout) :: var_list
        type(string_t), intent(in) :: name
        logical, intent(in), optional :: lval
        logical, intent(in), optional :: locked, verbose, intrinsic, user
        type(var_entry_t), pointer :: var
        allocate (var)
        call var_entry_init_log (var, name, lval, intrinsic, user)
        if (present (locked))  call var_entry_lock (var, locked)
        call var_list_append (var_list, var, verbose)
      end subroutine var_list_append_log_s

      subroutine var_list_append_int_s &
           (var_list, name, ival, locked, verbose, intrinsic, user)
        type(var_list_t), intent(inout) :: var_list
        type(string_t), intent(in) :: name
        integer, intent(in), optional :: ival
        logical, intent(in), optional :: locked, verbose, intrinsic, user
        type(var_entry_t), pointer :: var
        allocate (var)
        call var_entry_init_int (var, name, ival, intrinsic, user)
        if (present (locked))  call var_entry_lock (var, locked)
        call var_list_append (var_list, var, verbose)
      end subroutine var_list_append_int_s

      subroutine var_list_append_real_s &
           (var_list, name, rval, locked, verbose, intrinsic, user)
        type(var_list_t), intent(inout) :: var_list
        type(string_t), intent(in) :: name
        real(default), intent(in), optional :: rval
        logical, intent(in), optional :: locked, verbose, intrinsic, user
        type(var_entry_t), pointer :: var
        allocate (var)
        call var_entry_init_real (var, name, rval, intrinsic, user)
        if (present (locked))  call var_entry_lock (var, locked)
        call var_list_append (var_list, var, verbose)
      end subroutine var_list_append_real_s

      subroutine var_list_append_cmplx_s &
           (var_list, name, cval, locked, verbose, intrinsic, user)
        type(var_list_t), intent(inout) :: var_list
        type(string_t), intent(in) :: name
        complex(default), intent(in), optional :: cval
        logical, intent(in), optional :: locked, verbose, intrinsic, user
```

```
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_cmplx (var, name, cval, intrinsic, user)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_cmplx_s

subroutine var_list_append_subevt_s &
     (var_list, name, pval, locked, verbose, intrinsic, user)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  type(subevt_t), intent(in), optional :: pval
  logical, intent(in), optional :: locked, verbose, intrinsic, user
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_subevt (var, name, pval, intrinsic, user)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_subevt_s

subroutine var_list_append_pdg_array_s &
     (var_list, name, aval, locked, verbose, intrinsic, user)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  type(pdg_array_t), intent(in), optional :: aval
  logical, intent(in), optional :: locked, verbose, intrinsic, user
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_pdg_array (var, name, aval, intrinsic, user)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_pdg_array_s

subroutine var_list_append_string_s &
     (var_list, name, sval, locked, verbose, intrinsic, user)
  type(var_list_t), intent(inout) :: var_list
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: sval
  logical, intent(in), optional :: locked, verbose, intrinsic, user
  type(var_entry_t), pointer :: var
  allocate (var)
  call var_entry_init_string (var, name, sval, intrinsic, user)
  if (present (locked))  call var_entry_lock (var, locked)
  call var_list_append (var_list, var, verbose)
end subroutine var_list_append_string_s

subroutine var_list_append_log_c &
     (var_list, name, lval, locked, verbose, intrinsic, user)
  type(var_list_t), intent(inout) :: var_list
  character(*), intent(in) :: name
  logical, intent(in), optional :: lval
  logical, intent(in), optional :: locked, verbose, intrinsic, user
  call var_list_append_log_s &
       (var_list, var_str (name), lval, locked, verbose, intrinsic, user)
```

```
      end subroutine var_list_append_log_c

    subroutine var_list_append_int_c &
         (var_list, name, ival, locked, verbose, intrinsic, user)
      type(var_list_t), intent(inout) :: var_list
      character(*), intent(in) :: name
      integer, intent(in), optional :: ival
      logical, intent(in), optional :: locked, verbose, intrinsic, user
      call var_list_append_int_s &
           (var_list, var_str (name), ival, locked, verbose, intrinsic, user)
    end subroutine var_list_append_int_c

    subroutine var_list_append_real_c &
         (var_list, name, rval, locked, verbose, intrinsic, user)
      type(var_list_t), intent(inout) :: var_list
      character(*), intent(in) :: name
      real(default), intent(in), optional :: rval
      logical, intent(in), optional :: locked, verbose, intrinsic, user
      call var_list_append_real_s &
           (var_list, var_str (name), rval, locked, verbose, intrinsic, user)
    end subroutine var_list_append_real_c

    subroutine var_list_append_cmplx_c &
         (var_list, name, cval, locked, verbose, intrinsic, user)
      type(var_list_t), intent(inout) :: var_list
      character(*), intent(in) :: name
      complex(default), intent(in), optional :: cval
      logical, intent(in), optional :: locked, verbose, intrinsic, user
      call var_list_append_cmplx_s &
           (var_list, var_str (name), cval, locked, verbose, intrinsic, user)
    end subroutine var_list_append_cmplx_c

    subroutine var_list_append_subevt_c &
         (var_list, name, pval, locked, verbose, intrinsic, user)
      type(var_list_t), intent(inout) :: var_list
      character(*), intent(in) :: name
      type(subevt_t), intent(in), optional :: pval
      logical, intent(in), optional :: locked, verbose, intrinsic, user
      call var_list_append_subevt_s &
           (var_list, var_str (name), pval, locked, verbose, intrinsic, user)
    end subroutine var_list_append_subevt_c

    subroutine var_list_append_pdg_array_c &
         (var_list, name, aval, locked, verbose, intrinsic, user)
      type(var_list_t), intent(inout) :: var_list
      character(*), intent(in) :: name
      type(pdg_array_t), intent(in), optional :: aval
      logical, intent(in), optional :: locked, verbose, intrinsic, user
      call var_list_append_pdg_array_s &
           (var_list, var_str (name), aval, locked, verbose, intrinsic, user)
    end subroutine var_list_append_pdg_array_c

    subroutine var_list_append_string_c &
         (var_list, name, sval, locked, verbose, intrinsic, user)
```

```fortran
      type(var_list_t), intent(inout) :: var_list
      character(*), intent(in) :: name
      character(*), intent(in), optional :: sval
      logical, intent(in), optional :: locked, verbose, intrinsic, user
      if (present (sval)) then
         call var_list_append_string_s &
              (var_list, var_str (name), var_str (sval), &
              locked, verbose, intrinsic, user)
      else
         call var_list_append_string_s &
              (var_list, var_str (name), &
              locked=locked, verbose=verbose, intrinsic=intrinsic, user=user)
      end if
    end subroutine var_list_append_string_c
```

⟨*Variables: public*⟩+≡
```fortran
    public :: var_list_append_log_ptr
    public :: var_list_append_int_ptr
    public :: var_list_append_real_ptr
    public :: var_list_append_cmplx_ptr
    public :: var_list_append_pdg_array_ptr
    public :: var_list_append_subevt_ptr
    public :: var_list_append_string_ptr
```

⟨*Variables: procedures*⟩+≡
```fortran
    subroutine var_list_append_log_ptr &
         (var_list, name, lval, is_known, locked, verbose, intrinsic)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
      logical, intent(in), target :: lval
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: locked, verbose, intrinsic
      type(var_entry_t), pointer :: var
      allocate (var)
      call var_entry_init_log_ptr (var, name, lval, is_known, intrinsic)
      if (present (locked))  call var_entry_lock (var, locked)
      call var_list_append (var_list, var, verbose)
    end subroutine var_list_append_log_ptr

    subroutine var_list_append_int_ptr &
         (var_list, name, ival, is_known, locked, verbose, intrinsic)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in), target :: ival
      logical, intent(in), target :: is_known
      logical, intent(in), optional :: locked, verbose, intrinsic
      type(var_entry_t), pointer :: var
      allocate (var)
      call var_entry_init_int_ptr (var, name, ival, is_known, intrinsic)
      if (present (locked))  call var_entry_lock (var, locked)
      call var_list_append (var_list, var, verbose)
    end subroutine var_list_append_int_ptr

    subroutine var_list_append_real_ptr &
```

```
      (var_list, name, rval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    real(default), intent(in), target :: rval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_real_ptr (var, name, rval, is_known, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_real_ptr

  subroutine var_list_append_cmplx_ptr &
      (var_list, name, cval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    complex(default), intent(in), target :: cval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_cmplx_ptr (var, name, cval, is_known, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_cmplx_ptr

  subroutine var_list_append_pdg_array_ptr &
      (var_list, name, aval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_pdg_array_ptr (var, name, aval, is_known, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_pdg_array_ptr

  subroutine var_list_append_subevt_ptr &
      (var_list, name, pval, is_known, locked, verbose, intrinsic)
    type(var_list_t), intent(inout) :: var_list
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), target :: pval
    logical, intent(in), target :: is_known
    logical, intent(in), optional :: locked, verbose, intrinsic
    type(var_entry_t), pointer :: var
    allocate (var)
    call var_entry_init_subevt_ptr (var, name, pval, is_known, intrinsic)
    if (present (locked))  call var_entry_lock (var, locked)
    call var_list_append (var_list, var, verbose)
  end subroutine var_list_append_subevt_ptr
```

```
      subroutine var_list_append_string_ptr &
           (var_list, name, sval, is_known, locked, verbose, intrinsic)
        type(var_list_t), intent(inout) :: var_list
        type(string_t), intent(in) :: name
        type(string_t), intent(in), target :: sval
        logical, intent(in), target :: is_known
        logical, intent(in), optional :: locked, verbose, intrinsic
        type(var_entry_t), pointer :: var
        allocate (var)
        call var_entry_init_string_ptr (var, name, sval, is_known, intrinsic)
        if (present (locked))  call var_entry_lock (var, locked)
        call var_list_append (var_list, var, verbose)
      end subroutine var_list_append_string_ptr
```

Finalizer

Finalize, delete the list entry by entry. Follow link only if requested explicitly.

⟨*Variables: public*⟩+≡
```
    public :: var_list_final
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_list_final (var_list, follow_link)
      type(var_list_t), intent(inout) :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var_list%last => null ()
      do while (associated (var_list%first))
         var => var_list%first
         var_list%first => var%next
         call var_entry_final (var)
         deallocate (var)
      end do
      if (present (follow_link)) then
         if (follow_link .and. associated (var_list%next)) then
            call var_list_final (var_list%next, follow_link)
            deallocate (var_list%next)
         end if
      end if
    end subroutine var_list_final
```

Output

Optionally, show only variables of a certain type.

⟨*Variables: public*⟩+≡
```
    public :: var_list_write
```

⟨*Variables: procedures*⟩+≡
```
    recursive subroutine var_list_write &
         (var_list, unit, follow_link, only_type, prefix, model_name, show_ptr, &
          intrinsic, pacified)
      type(var_list_t), intent(in), target :: var_list
```

```fortran
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: follow_link
      integer, intent(in), optional :: only_type
      character(*), intent(in), optional :: prefix
      type(string_t), intent(in), optional :: model_name
      logical, intent(in), optional :: show_ptr
      logical, intent(in), optional :: intrinsic
      logical, intent(in), optional :: pacified
      type(var_entry_t), pointer :: var
      integer :: u, length
      logical :: write_this, write_next
      u = output_unit (unit);  if (u < 0)  return
      if (present (prefix))  length = len (prefix)
      var => var_list%first
      if (associated (var)) then
         do while (associated (var))
            if (present (only_type)) then
               write_this = only_type == var%type
            else
               write_this = .true.
            end if
            if (write_this .and. present (prefix)) then
               if (prefix /= extract (var%name, 1, length)) &
                    write_this = .false.
            end if
            if (write_this) then
               call var_entry_write &
                    (var, unit, model_name = model_name, show_ptr = show_ptr, &
                     intrinsic = intrinsic, pacified = pacified)
            end if
            var => var%next
         end do
      end if
      write_next = associated (var_list%next)
      if (present (follow_link)) &
           write_next = write_next .and. follow_link
      if (write_next) then
         call var_list_write (var_list%next, &
              unit, follow_link, only_type, prefix, model_name, show_ptr, &
              intrinsic, pacified)
      end if
    end subroutine var_list_write
```

Write only a certain variable.

⟨Variables: public⟩+≡
```fortran
    public :: var_list_write_var
```

⟨Variables: procedures⟩+≡
```fortran
    recursive subroutine var_list_write_var &
        (var_list, name, unit, type, follow_link, &
         model_name, show_ptr, pacified)
      type(var_list_t), intent(in), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: unit
```

```
      integer, intent(in), optional :: type
      logical, intent(in), optional :: follow_link
      type(string_t), intent(in), optional :: model_name
      logical, intent(in), optional :: show_ptr
      logical, intent(in), optional :: pacified
      type(var_entry_t), pointer :: var
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      var => var_list_get_var_ptr &
           (var_list, name, type, follow_link=follow_link, defined=.true.)
      if (associated (var)) then
         call var_entry_write &
              (var, unit, model_name = model_name, &
              show_ptr = show_ptr, pacified = pacified)
      else
         write (u, "(A)")  char (name) // " = [undefined]"
      end if
    end subroutine var_list_write_var
```

5.6.5 Tools

Return a pointer to the variable list linked to by the current one.

⟨*Variables: public*⟩+≡
```
    public :: var_list_get_next_ptr
```

⟨*Variables: procedures*⟩+≡
```
    function var_list_get_next_ptr (var_list) result (next_ptr)
      type(var_list_t), pointer :: next_ptr
      type(var_list_t), intent(in) :: var_list
      next_ptr => var_list%next
    end function var_list_get_next_ptr
```

Return a pointer to the variable with the requested name. If no such name
exists, return a null pointer. In that case, try the next list if present, unless
`follow_link` is unset. If `defined` is set, ignore entries that exist but are unde-
fined.

⟨*Variables: public*⟩+≡
```
    public :: var_list_get_var_ptr
```

⟨*Variables: procedures*⟩+≡
```
    recursive function var_list_get_var_ptr &
         (var_list, name, type, follow_link, defined) result (var)
      type(var_entry_t), pointer :: var
      type(var_list_t), intent(in), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: type
      logical, intent(in), optional :: follow_link, defined
      logical :: ignore_undef, search_next
      ignore_undef = .true.;  if (present (defined))  ignore_undef = .not. defined
      var => var_list%first
      if (present (type)) then
         do while (associated (var))
```

```
               if (var%type == type) then
                  if (var%name == name) then
                     if (ignore_undef .or. var%is_defined)  return
                  end if
               end if
               var => var%next
            end do
         else
            do while (associated (var))
               if (var%name == name) then
                  if (ignore_undef .or. var%is_defined)  return
               end if
               var => var%next
            end do
         end if
         search_next = associated (var_list%next)
         if (present (follow_link)) &
              search_next = search_next .and. follow_link
         if (search_next) &
              var => var_list_get_var_ptr &
                   (var_list%next, name, type, defined=defined)
      end function var_list_get_var_ptr
```

Return the variable type

⟨*Variables: public*⟩+≡
```
      public :: var_list_get_type
```

⟨*Variables: procedures*⟩+≡
```
      function var_list_get_type (var_list, name, follow_link) result (type)
        integer :: type
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
        if (associated (var)) then
           type = var%type
        else
           type = V_NONE
        end if
      end function var_list_get_type
```

Return true if the variable exists.

⟨*Variables: public*⟩+≡
```
      public :: var_list_exists
```

⟨*Variables: procedures*⟩+≡
```
      function var_list_exists (var_list, name, follow_link) result (flag)
        logical :: flag
        type(string_t), intent(in) :: name
        type(var_list_t), intent(in), target :: var_list
        logical, intent(in), optional :: follow_link
        type(var_entry_t), pointer :: var
        var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
```

```
      flag = associated (var)
   end function var_list_exists
```

Return true if the variable is declared as intrinsic.

⟨*Variables: public*⟩+≡
```
    public :: var_list_is_intrinsic
```

⟨*Variables: procedures*⟩+≡
```
    function var_list_is_intrinsic (var_list, name, follow_link) result (flag)
      logical :: flag
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
      if (associated (var)) then
         flag = var%is_intrinsic
      else
         flag = .false.
      end if
    end function var_list_is_intrinsic
```

Return true if the value is known.

⟨*Variables: public*⟩+≡
```
    public :: var_list_is_known
```

⟨*Variables: interfaces*⟩+≡
```
    interface var_list_is_known
       module procedure var_list_is_known_s
       module procedure var_list_is_known_c
    end interface
```

⟨*Variables: procedures*⟩+≡
```
    function var_list_is_known_s (var_list, name, follow_link) result (flag)
      logical :: flag
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
      if (associated (var)) then
         flag = var%is_known
      else
         flag = .false.
      end if
    end function var_list_is_known_s

    function var_list_is_known_c (var_list, name, follow_link) result (flag)
      logical :: flag
      character(*), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      flag = var_list_is_known_s (var_list, var_str (name), follow_link)
    end function var_list_is_known_c
```

Return true if the value is locked.

⟨Variables: public⟩+≡
```
    public :: var_list_is_locked
```

⟨Variables: procedures⟩+≡
```
    function var_list_is_locked (var_list, name, follow_link) result (flag)
      logical :: flag
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, follow_link=follow_link)
      if (associated (var)) then
         flag = var_entry_is_locked (var)
      else
         flag = .false.
      end if
    end function var_list_is_locked
```

Return the value, assuming that the type is correct. We consider only variable
entries that have been `defined`.

For convenience, allow both variable and fixed-length (literal) strings.

⟨Variables: public⟩+≡
```
    public :: var_list_get_lval
    public :: var_list_get_ival
    public :: var_list_get_rval
    public :: var_list_get_cval
    public :: var_list_get_pval
    public :: var_list_get_aval
    public :: var_list_get_sval
```

⟨Variables: interfaces⟩+≡
```
    interface var_list_get_lval
       module procedure var_list_get_lval_s
       module procedure var_list_get_lval_c
    end interface
    interface var_list_get_ival
       module procedure var_list_get_ival_s
       module procedure var_list_get_ival_c
    end interface
    interface var_list_get_rval
       module procedure var_list_get_rval_s
       module procedure var_list_get_rval_c
    end interface
    interface var_list_get_cval
       module procedure var_list_get_cval_s
       module procedure var_list_get_cval_c
    end interface
    interface var_list_get_pval
       module procedure var_list_get_pval_s
       module procedure var_list_get_pval_c
    end interface
    interface var_list_get_aval
       module procedure var_list_get_aval_s
```

```
      module procedure var_list_get_aval_c
    end interface
    interface var_list_get_sval
      module procedure var_list_get_sval_s
      module procedure var_list_get_sval_c
    end interface
```

⟨*Variables: procedures*⟩+≡

```
    function var_list_get_lval_s (var_list, name, follow_link) result (lval)
      logical :: lval
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr &
           (var_list, name, V_LOG, follow_link, defined=.true.)
      if (associated (var)) then
         if (var_has_value (var)) then
            lval = var%lval
         else
            lval = .false.
         end if
      else
         lval = .false.
      end if
    end function var_list_get_lval_s

    function var_list_get_ival_s (var_list, name, follow_link) result (ival)
      integer :: ival
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr &
           (var_list, name, V_INT, follow_link, defined=.true.)
      if (associated (var)) then
         if (var_has_value (var)) then
            ival = var%ival
         else
            ival = 0
         end if
      else
         ival = 0
      end if
    end function var_list_get_ival_s

    function var_list_get_rval_s (var_list, name, follow_link) result (rval)
      real(default) :: rval
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr &
           (var_list, name, V_REAL, follow_link, defined=.true.)
```

```
    if (associated (var)) then
       if (var_has_value (var)) then
          rval = var%rval
       else
          rval = 0
       end if
    else
       rval = 0
    end if
end function var_list_get_rval_s

function var_list_get_cval_s (var_list, name, follow_link) result (cval)
  complex(default) :: cval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr &
       (var_list, name, V_CMPLX, follow_link, defined=.true.)
  if (associated (var)) then
     if (var_has_value (var)) then
        cval = var%cval
     else
        cval = 0
     end if
  else
     cval = 0
  end if
end function var_list_get_cval_s

function var_list_get_aval_s (var_list, name, follow_link) result (aval)
  type(pdg_array_t) :: aval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr &
       (var_list, name, V_PDG, follow_link, defined=.true.)
  if (associated (var)) then
     if (var_has_value (var)) then
        aval = var%aval
     end if
  end if
end function var_list_get_aval_s

function var_list_get_pval_s (var_list, name, follow_link) result (pval)
  type(subevt_t) :: pval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr &
       (var_list, name, V_SEV, follow_link, defined=.true.)
  if (associated (var)) then
```

```
      if (var_has_value (var)) then
         pval = var%pval
      end if
   end if
end function var_list_get_pval_s

function var_list_get_sval_s (var_list, name, follow_link) result (sval)
  type(string_t) :: sval
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr &
       (var_list, name, V_STR, follow_link, defined=.true.)
  if (associated (var)) then
     if (var_has_value (var)) then
        sval = var%sval
     else
        sval = ""
     end if
  else
     sval = ""
  end if
end function var_list_get_sval_s

function var_list_get_lval_c (var_list, name, follow_link) result (lval)
  logical :: lval
  character(*), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  lval = var_list_get_lval_s (var_list, var_str (name), follow_link)
end function var_list_get_lval_c

function var_list_get_ival_c (var_list, name, follow_link) result (ival)
  integer :: ival
  character(*), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  ival = var_list_get_ival_s (var_list, var_str (name), follow_link)
end function var_list_get_ival_c

function var_list_get_rval_c (var_list, name, follow_link) result (rval)
  real(default) :: rval
  character(*), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
  rval = var_list_get_rval_s (var_list, var_str (name), follow_link)
end function var_list_get_rval_c

function var_list_get_cval_c (var_list, name, follow_link) result (cval)
  complex(default) :: cval
  character(*), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  logical, intent(in), optional :: follow_link
```

```
      cval = var_list_get_cval_s (var_list, var_str (name), follow_link)
    end function var_list_get_cval_c

    function var_list_get_aval_c (var_list, name, follow_link) result (aval)
      type(pdg_array_t) :: aval
      character(*), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      aval = var_list_get_aval_s (var_list, var_str (name), follow_link)
    end function var_list_get_aval_c

    function var_list_get_pval_c (var_list, name, follow_link) result (pval)
      type(subevt_t) :: pval
      character(*), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      pval = var_list_get_pval_s (var_list, var_str (name), follow_link)
    end function var_list_get_pval_c

    function var_list_get_sval_c (var_list, name, follow_link) result (sval)
      type(string_t) :: sval
      character(*), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      logical, intent(in), optional :: follow_link
      sval = var_list_get_sval_s (var_list, var_str (name), follow_link)
    end function var_list_get_sval_c
```

Check for a valid value, given a pointer. Issue error messages if invalid.

⟨*Variables: procedures*⟩+≡

```
    function var_has_value (var) result (valid)
      logical :: valid
      type(var_entry_t), pointer :: var
      if (associated (var)) then
         if (var%is_known) then
            valid = .true.
         else
            call msg_error ("The value of variable '" // char (var%name) &
                 // "' is unknown but must be known at this point.")
            valid = .false.
         end if
      else
         call msg_error ("Variable '" // char (var%name) &
              // "' is undefined but must have a known value at this point.")
         valid = .false.
      end if
    end function var_has_value
```

5.6.6 Process-specific variables

We allow the user to set a numeric process ID for each declared process.

⟨*Variables: public*⟩+≡

```
    public :: var_list_init_num_id
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_init_num_id (var_list, proc_id, num_id)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: proc_id
      integer, intent(in), optional :: num_id
      call var_list_set_procvar_int (var_list, proc_id, &
           var_str ("num_id"), num_id)
    end subroutine var_list_init_num_id
```

Integration results are stored in special variables. They are initialized by this subroutine. The values may or may not already known.

Note: the values which are accessible are those that are unique for a process with multiple MCI records. The rest has been discarded.

⟨*Variables: public*⟩+≡
```
    public :: var_list_init_process_results
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_init_process_results (var_list, proc_id, &
         n_calls, integral, error, accuracy, chi2, efficiency)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: proc_id
      integer, intent(in), optional :: n_calls
      real(default), intent(in), optional :: integral, error, accuracy
      real(default), intent(in), optional :: chi2, efficiency
      call var_list_set_procvar_real (var_list, proc_id, &
           var_str ("integral"), integral)
      call var_list_set_procvar_real (var_list, proc_id, &
           var_str ("error"), error)
    end subroutine var_list_init_process_results
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_set_procvar_int (var_list, proc_id, name, ival)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: proc_id
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: ival
      type(string_t) :: var_name
      type(var_entry_t), pointer :: var
      var_name = name // "(" // proc_id // ")"
      var => var_list_get_var_ptr (var_list, var_name)
      if (.not. associated (var)) then
         call var_list_append_int (var_list, var_name, ival, intrinsic=.true.)
      else if (present (ival)) then
         call var_list_set_int (var_list, var_name, ival, is_known=.true.)
      end if
    end subroutine var_list_set_procvar_int

    subroutine var_list_set_procvar_real (var_list, proc_id, name, rval)
      type(var_list_t), intent(inout) :: var_list
      type(string_t), intent(in) :: proc_id
      type(string_t), intent(in) :: name
      real(default), intent(in), optional :: rval
      type(string_t) :: var_name
```

```
      type(var_entry_t), pointer :: var
      var_name = name // "(" // proc_id // ")"
      var => var_list_get_var_ptr (var_list, var_name)
      if (.not. associated (var)) then
         call var_list_append_real (var_list, var_name, rval, intrinsic=.true.)
      else if (present (rval)) then
         call var_list_set_real (var_list, var_name, rval, is_known=.true.)
      end if
   end subroutine var_list_set_procvar_real
```

5.6.7   Observable initialization

Observables are formally treated as variables, which however are evaluated each time the observable is used. The arguments (pointers) to evaluate and the function are part of the variable-list entry.

The procedure pointer should be set by this subroutine. This, however, triggers a bug in nagfor 5.2(649). As a workaround, we return the variable pointer, so the pointer can be set directly.

⟨*Variables: procedures*⟩+≡
```
   subroutine var_list_set_obs (var_list, name, type, var, prt1, prt2)
     type(var_list_t), intent(inout) :: var_list
     type(string_t), intent(in) :: name
     integer, intent(in) :: type
     type(var_entry_t), pointer :: var
     type(prt_t), intent(in), target :: prt1
     type(prt_t), intent(in), optional, target :: prt2
     allocate (var)
     call var_entry_init_obs (var, name, type, prt1, prt2)
     call var_list_append (var_list, var)
   end subroutine var_list_set_obs
```

Unary and binary observables are different. Most unary observables can be equally well evaluated for particle pairs. Binary observables cannot be evaluated for single particles.

⟨*Variables: public*⟩+≡
```
   public :: var_list_set_observables_unary
   public :: var_list_set_observables_binary
```

⟨*Variables: procedures*⟩+≡
```
   subroutine var_list_set_observables_unary (var_list, prt1)
     type(var_list_t), intent(inout) :: var_list
     type(prt_t), intent(in), target :: prt1
     type(var_entry_t), pointer :: var
     call var_list_set_obs &
          (var_list, var_str ("PDG"), V_OBS1_INT, var, prt1)
     var% obs1_int => obs_pdg1
     call var_list_set_obs &
          (var_list, var_str ("Hel"), V_OBS1_INT, var, prt1)
     var% obs1_int => obs_helicity1
     call var_list_set_obs &
          (var_list, var_str ("M"), V_OBS1_REAL, var, prt1)
     var% obs1_real => obs_signed_mass1
```

```
    call var_list_set_obs &
         (var_list, var_str ("M2"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_mass_squared1
    call var_list_set_obs &
         (var_list, var_str ("E"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_energy1
    call var_list_set_obs &
         (var_list, var_str ("Px"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_px1
    call var_list_set_obs &
         (var_list, var_str ("Py"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_py1
    call var_list_set_obs &
         (var_list, var_str ("Pz"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_pz1
    call var_list_set_obs &
         (var_list, var_str ("P"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_p1
    call var_list_set_obs &
         (var_list, var_str ("Pl"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_pl1
    call var_list_set_obs &
         (var_list, var_str ("Pt"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_pt1
    call var_list_set_obs &
         (var_list, var_str ("Theta"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_theta1
    call var_list_set_obs &
         (var_list, var_str ("Phi"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_phi1
    call var_list_set_obs &
         (var_list, var_str ("Rap"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_rap1
    call var_list_set_obs &
         (var_list, var_str ("Eta"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_eta1
    call var_list_set_obs &
         (var_list, var_str ("Theta_RF"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_theta_rf1
    call var_list_set_obs &
         (var_list, var_str ("Dist"), V_OBS1_REAL, var, prt1)
    var% obs1_real => obs_dist1
    call var_list_set_obs &
         (var_list, var_str ("_User_obs_real"), V_UOBS1_REAL, var, prt1)
    call var_list_set_obs &
         (var_list, var_str ("_User_obs_int"), V_UOBS1_INT, var, prt1)
end subroutine var_list_set_observables_unary

subroutine var_list_set_observables_binary (var_list, prt1, prt2)
  type(var_list_t), intent(inout) :: var_list
  type(prt_t), intent(in), target :: prt1
  type(prt_t), intent(in), optional, target :: prt2
  type(var_entry_t), pointer :: var
  call var_list_set_obs &
```

```
      (var_list, var_str ("PDG"), V_OBS2_INT, var, prt1, prt2)
var% obs2_int => obs_pdg2
call var_list_set_obs &
      (var_list, var_str ("Hel"), V_OBS2_INT, var, prt1, prt2)
var% obs2_int => obs_helicity2
call var_list_set_obs &
      (var_list, var_str ("M"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_signed_mass2
call var_list_set_obs &
      (var_list, var_str ("M2"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_mass_squared2
call var_list_set_obs &
      (var_list, var_str ("E"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_energy2
call var_list_set_obs &
      (var_list, var_str ("Px"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_px2
call var_list_set_obs &
      (var_list, var_str ("Py"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_py2
call var_list_set_obs &
      (var_list, var_str ("Pz"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_pz2
call var_list_set_obs &
      (var_list, var_str ("P"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_p2
call var_list_set_obs &
      (var_list, var_str ("Pl"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_pl2
call var_list_set_obs &
      (var_list, var_str ("Pt"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_pt2
call var_list_set_obs &
      (var_list, var_str ("Theta"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_theta2
call var_list_set_obs &
      (var_list, var_str ("Phi"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_phi2
call var_list_set_obs &
      (var_list, var_str ("Rap"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_rap2
call var_list_set_obs &
      (var_list, var_str ("Eta"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_eta2
call var_list_set_obs &
      (var_list, var_str ("Theta_RF"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_theta_rf2
call var_list_set_obs &
      (var_list, var_str ("Dist"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_dist2
call var_list_set_obs &
      (var_list, var_str ("kT"), V_OBS2_REAL, var, prt1, prt2)
var% obs2_real => obs_ktmeasure
call var_list_set_obs &
```

```
          (var_list, var_str ("_User_obs_real"), V_UOBS2_REAL, var, prt1, prt2)
    call var_list_set_obs &
          (var_list, var_str ("_User_obs_int"), V_UOBS2_INT, var, prt1, prt2)
  end subroutine var_list_set_observables_binary
```

Check if a variable name is defined as an observable:

⟨*Variables: procedures*⟩+≡
```
  function string_is_observable_id (string) result (flag)
    logical :: flag
    type(string_t), intent(in) :: string
    select case (char (string))
    case ("PDG", "Hel", "M", "M2", "E", "Px", "Py", "Pz", "P", "Pl", "Pt", &
          "Theta", "Phi", "Rap", "Eta", "Theta_RF", "Dist", "kT")
       flag = .true.
    case default
       flag = .false.
    end select
  end function string_is_observable_id
```

5.6.8   Observables

These are analogous to the unary and binary numeric functions listed above.
An observable takes the `pval` component(s) of its one or two argument nodes
and produces an integer or real value.

Integer-valued unary observables

The PDG code

⟨*Variables: procedures*⟩+≡
```
  integer function obs_pdg1 (prt1) result (pdg)
    type(prt_t), intent(in) :: prt1
    pdg = prt_get_pdg (prt1)
  end function obs_pdg1
```

The helicity. The return value is meaningful only if the particle is polarized,
otherwise an invalid value is returned (-9).

⟨*Variables: procedures*⟩+≡
```
  integer function obs_helicity1 (prt1) result (h)
    type(prt_t), intent(in) :: prt1
    if (prt_is_polarized (prt1)) then
       h = prt_get_helicity (prt1)
    else
       h = -9
    end if
  end function obs_helicity1
```

Real-valued unary observables

The invariant mass squared, obtained from the separately stored value.

⟨*Variables: procedures*⟩+≡

```
    real(default) function obs_mass_squared1 (prt1) result (p2)
      type(prt_t), intent(in) :: prt1
      p2 = prt_get_msq (prt1)
    end function obs_mass_squared1
```

The signed invariant mass, which is the signed square root of the previous observable.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_signed_mass1 (prt1) result (m)
      type(prt_t), intent(in) :: prt1
      real(default) :: msq
      msq = prt_get_msq (prt1)
      m = sign (sqrt (abs (msq)), msq)
    end function obs_signed_mass1
```

The particle energy

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_energy1 (prt1) result (e)
      type(prt_t), intent(in) :: prt1
      e = energy (prt_get_momentum (prt1))
    end function obs_energy1
```

Particle momentum (components)

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_px1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = vector4_get_component (prt_get_momentum (prt1), 1)
    end function obs_px1

    real(default) function obs_py1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = vector4_get_component (prt_get_momentum (prt1), 2)
    end function obs_py1

    real(default) function obs_pz1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = vector4_get_component (prt_get_momentum (prt1), 3)
    end function obs_pz1

    real(default) function obs_p1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = space_part_norm (prt_get_momentum (prt1))
    end function obs_p1

    real(default) function obs_pl1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = longitudinal_part (prt_get_momentum (prt1))
    end function obs_pl1

    real(default) function obs_pt1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = transverse_part (prt_get_momentum (prt1))
```

```
        end function obs_pt1
```

Polar and azimuthal angle (lab frame).

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_theta1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = polar_angle (prt_get_momentum (prt1))
    end function obs_theta1

    real(default) function obs_phi1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = azimuthal_angle (prt_get_momentum (prt1))
    end function obs_phi1
```

Rapidity and pseudorapidity

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_rap1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = rapidity (prt_get_momentum (prt1))
    end function obs_rap1

    real(default) function obs_eta1 (prt1) result (p)
      type(prt_t), intent(in) :: prt1
      p = pseudorapidity (prt_get_momentum (prt1))
    end function obs_eta1
```

Meaningless: Polar angle in the rest frame of the 2nd argument.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_theta_rf1 (prt1) result (dist)
      type(prt_t), intent(in) :: prt1
      call msg_fatal (" 'Theta_RF' is undefined as unary observable")
      dist = 0
    end function obs_theta_rf1
```

Meaningless: Distance on the η-ϕ cylinder.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_dist1 (prt1) result (dist)
      type(prt_t), intent(in) :: prt1
      call msg_fatal (" 'Dist' is undefined as unary observable")
      dist = 0
    end function obs_dist1
```

Integer-valued binary observables

These observables are meaningless as binary functions.

⟨*Variables: procedures*⟩+≡
```
    integer function obs_pdg2 (prt1, prt2) result (pdg)
      type(prt_t), intent(in) :: prt1, prt2
      call msg_fatal (" PDG_Code is undefined as binary observable")
      pdg = 0
```

```
    end function obs_pdg2

    integer function obs_helicity2 (prt1, prt2) result (h)
      type(prt_t), intent(in) :: prt1, prt2
      call msg_fatal (" Helicity is undefined as binary observable")
      h = 0
    end function obs_helicity2
```

Real-valued binary observables

The invariant mass squared, obtained from the separately stored value.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_mass_squared2 (prt1, prt2) result (p2)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p2 = prt_get_msq (prt)
    end function obs_mass_squared2
```

The signed invariant mass, which is the signed square root of the previous observable.

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_signed_mass2 (prt1, prt2) result (m)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      real(default) :: msq
      call prt_init_combine (prt, prt1, prt2)
      msq = prt_get_msq (prt)
      m = sign (sqrt (abs (msq)), msq)
    end function obs_signed_mass2
```

The particle energy

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_energy2 (prt1, prt2) result (e)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      e = energy (prt_get_momentum (prt))
    end function obs_energy2
```

Particle momentum (components)

⟨*Variables: procedures*⟩+≡
```
    real(default) function obs_px2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = vector4_get_component (prt_get_momentum (prt), 1)
    end function obs_px2

    real(default) function obs_py2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
```

```
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = vector4_get_component (prt_get_momentum (prt), 2)
    end function obs_py2

    real(default) function obs_pz2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = vector4_get_component (prt_get_momentum (prt), 3)
    end function obs_pz2

    real(default) function obs_p2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = space_part_norm (prt_get_momentum (prt))
    end function obs_p2

    real(default) function obs_pl2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = longitudinal_part (prt_get_momentum (prt))
    end function obs_pl2

    real(default) function obs_pt2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = transverse_part (prt_get_momentum (prt))
    end function obs_pt2
```

Enclosed angle and azimuthal distance (lab frame).

⟨*Variables: procedures*⟩+≡

```
    real(default) function obs_theta2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      p = enclosed_angle (prt_get_momentum (prt1), prt_get_momentum (prt2))
    end function obs_theta2

    real(default) function obs_phi2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      type(prt_t) :: prt
      call prt_init_combine (prt, prt1, prt2)
      p = azimuthal_distance (prt_get_momentum (prt1), prt_get_momentum (prt2))
    end function obs_phi2
```

Rapidity and pseudorapidity distance

⟨*Variables: procedures*⟩+≡

```
    real(default) function obs_rap2 (prt1, prt2) result (p)
      type(prt_t), intent(in) :: prt1, prt2
      p = rapidity_distance &
          (prt_get_momentum (prt1), prt_get_momentum (prt2))
```

```
      end function obs_rap2

      real(default) function obs_eta2 (prt1, prt2) result (p)
        type(prt_t), intent(in) :: prt1, prt2
        type(prt_t) :: prt
        call prt_init_combine (prt, prt1, prt2)
        p = pseudorapidity_distance &
             (prt_get_momentum (prt1), prt_get_momentum (prt2))
      end function obs_eta2
```

Polar angle in the rest frame of the 2nd argument.

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_theta_rf2 (prt1, prt2) result (theta)
        type(prt_t), intent(in) :: prt1, prt2
        theta = enclosed_angle_rest_frame &
             (prt_get_momentum (prt1), prt_get_momentum (prt2))
      end function obs_theta_rf2
```

Distance on the η-ϕ cylinder.

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_dist2 (prt1, prt2) result (dist)
        type(prt_t), intent(in) :: prt1, prt2
        dist = eta_phi_distance &
             (prt_get_momentum (prt1), prt_get_momentum (prt2))
      end function obs_dist2
```

Durham kT measure.

⟨*Variables: procedures*⟩+≡
```
      real(default) function obs_ktmeasure (prt1, prt2) result (kt)
        type(prt_t), intent(in) :: prt1, prt2
        real (default) :: q2, e1, e2
        ! Normalized scale to one for now! (#67)
        q2 = 1
        e1 = energy (prt_get_momentum (prt1))
        e2 = energy (prt_get_momentum (prt2))
        kt = (2/q2) * min(e1**2,e2**2) *  &
             (1 - enclosed_angle_ct(prt_get_momentum (prt1), &
             prt_get_momentum (prt2)))
      end function obs_ktmeasure
```

5.6.9   API for variable lists

Set a new value. If the variable holds a pointer, this pointer is followed, e.g.,
a model parameter is actually set. If `ignore` is set, do nothing if the variable
does not exist. If `verbose` is set, echo the new value.

   The specific versions:

⟨*Variables: public*⟩+≡
```
      public :: var_list_set_log
      public :: var_list_set_int
      public :: var_list_set_real
```

```
      public :: var_list_set_cmplx
      public :: var_list_set_subevt
      public :: var_list_set_pdg_array
      public :: var_list_set_string
⟨Variables: procedures⟩+≡
    subroutine var_list_set_log &
         (var_list, name, lval, is_known, ignore, verbose, model_name)
      type(var_list_t), intent(inout), target :: var_list
      type(string_t), intent(in) :: name
      logical, intent(in) :: lval
      logical, intent(in) :: is_known
      logical, intent(in), optional :: ignore, verbose
      type(string_t), intent(in), optional :: model_name
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, V_LOG)
      if (associated (var)) then
         if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_LOG)
               call var_entry_set_log (var, lval, is_known, verbose, model_name)
            case default
               call var_mismatch_error (name)
            end select
         else
            call var_locked_error (name)
         end if
      else
         call var_missing_error (name, ignore)
      end if
    end subroutine var_list_set_log

    subroutine var_list_set_int &
         (var_list, name, ival, is_known, ignore, verbose, model_name)
      type(var_list_t), intent(inout), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(in) :: ival
      logical, intent(in) :: is_known
      logical, intent(in), optional :: ignore, verbose
      type(string_t), intent(in), optional :: model_name
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, V_INT)
      if (associated (var)) then
         if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_INT)
               call var_entry_set_int (var, ival, is_known, verbose, model_name)
            case default
               call var_mismatch_error (name)
            end select
         else
            call var_locked_error (name)
         end if
      else
         call var_missing_error (name, ignore)
```

```fortran
    end if
end subroutine var_list_set_int

subroutine var_list_set_real &
     (var_list, name, rval, is_known, ignore, &
      verbose, model_name, pacified)
  type(var_list_t), intent(inout), target :: var_list
  type(string_t), intent(in) :: name
  real(default), intent(in) :: rval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: ignore, verbose, pacified
  type(string_t), intent(in), optional :: model_name
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_REAL)
  if (associated (var)) then
     if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_REAL)
           call var_entry_set_real &
                (var, rval, is_known, verbose, model_name, pacified)
        case default
           call var_mismatch_error (name)
        end select
     else
        call var_locked_error (name)
     end if
  else
     call var_missing_error (name, ignore)
  end if
end subroutine var_list_set_real

subroutine var_list_set_cmplx &
     (var_list, name, cval, is_known, ignore, &
      verbose, model_name, pacified)
  type(var_list_t), intent(inout), target :: var_list
  type(string_t), intent(in) :: name
  complex(default), intent(in) :: cval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: ignore, verbose, pacified
  type(string_t), intent(in), optional :: model_name
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_CMPLX)
  if (associated (var)) then
     if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_CMPLX)
           call var_entry_set_cmplx &
                (var, cval, is_known, verbose, model_name, pacified)
        case default
           call var_mismatch_error (name)
        end select
     else
        call var_locked_error (name)
     end if
```

```
      else
         call var_missing_error (name, ignore)
      end if
    end subroutine var_list_set_cmplx

    subroutine var_list_set_pdg_array &
         (var_list, name, aval, is_known, ignore, verbose, model_name)
      type(var_list_t), intent(inout), target :: var_list
      type(string_t), intent(in) :: name
      type(pdg_array_t), intent(in) :: aval
      logical, intent(in) :: is_known
      logical, intent(in), optional :: ignore, verbose
      type(string_t), intent(in), optional :: model_name
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, V_PDG)
      if (associated (var)) then
         if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_PDG)
               call var_entry_set_pdg_array &
                    (var, aval, is_known, verbose, model_name)
            case default
               call var_mismatch_error (name)
            end select
         else
            call var_locked_error (name)
         end if
      else
         call var_missing_error (name, ignore)
      end if
    end subroutine var_list_set_pdg_array

    subroutine var_list_set_subevt &
         (var_list, name, pval, is_known, ignore, verbose, model_name)
      type(var_list_t), intent(inout), target :: var_list
      type(string_t), intent(in) :: name
      type(subevt_t), intent(in) :: pval
      logical, intent(in) :: is_known
      logical, intent(in), optional :: ignore, verbose
      type(string_t), intent(in), optional :: model_name
      type(var_entry_t), pointer :: var
      var => var_list_get_var_ptr (var_list, name, V_SEV)
      if (associated (var)) then
         if (.not. var_entry_is_locked (var)) then
            select case (var%type)
            case (V_SEV)
               call var_entry_set_subevt &
                    (var, pval, is_known, verbose, model_name)
            case default
               call var_mismatch_error (name)
            end select
         else
            call var_locked_error (name)
         end if
```

```fortran
      else
         call var_missing_error (name, ignore)
      end if
end subroutine var_list_set_subevt

subroutine var_list_set_string &
     (var_list, name, sval, is_known, ignore, verbose, model_name)
  type(var_list_t), intent(inout), target :: var_list
  type(string_t), intent(in) :: name
  type(string_t), intent(in) :: sval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: ignore, verbose
  type(string_t), intent(in), optional :: model_name
  type(var_entry_t), pointer :: var
  var => var_list_get_var_ptr (var_list, name, V_STR)
  if (associated (var)) then
     if (.not. var_entry_is_locked (var)) then
        select case (var%type)
        case (V_STR)
           call var_entry_set_string &
                (var, sval, is_known, verbose, model_name)
        case default
           call var_mismatch_error (name)
        end select
     else
        call var_locked_error (name)
     end if
  else
     call var_missing_error (name, ignore)
  end if
end subroutine var_list_set_string

subroutine var_mismatch_error (name)
  type(string_t), intent(in) :: name
  call msg_fatal ("Type mismatch for variable '" // char (name) // "'")
end subroutine var_mismatch_error

subroutine var_locked_error (name)
  type(string_t), intent(in) :: name
  call msg_error ("Variable '" // char (name) // "' is not user-definable")
end subroutine var_locked_error

subroutine var_missing_error (name, ignore)
  type(string_t), intent(in) :: name
  logical, intent(in), optional :: ignore
  logical :: error
  if (present (ignore)) then
     error = .not. ignore
  else
     error = .true.
  end if
  if (error) then
     call msg_fatal ("Variable '" // char (name) // "' has not been declared")
  end if
```

```
      end subroutine var_missing_error
```

5.6.10   Linking model variables

The variable list of a model can be linked to the global variable list, so the
model variables become available. However, the model may change during the
execution of the command list, and this is not known at compile time. So, we
make a copy of all variables that can be modified by the user; this will include
all variables that are present in any of the models. At runtime, the linked list
and the pointers to it can be exchanged, but the global variables will stay.

Append a single model variable to the local variable list. The pointer to the
original is not set (yet). Check first if it already exists; in that case, do nothing.

⟨*Variables: public*⟩+≡
```
      public :: var_list_init_copy
```

⟨*Variables: procedures*⟩+≡
```
      subroutine var_list_init_copy (var_list, model_var, user)
        type(var_list_t), intent(inout), target :: var_list
        type(var_entry_t), intent(in), target :: model_var
        logical, intent(in), optional :: user
        type(var_entry_t), pointer :: var
        if (.not. var_list_exists &
             (var_list, model_var%name, follow_link = .false.)) then
           allocate (var)
           call var_entry_init_copy (var, model_var, user)
           call var_list_append (var_list, var)
        end if
      end subroutine var_list_init_copy
```

Append all model variables, or reuse (unset) them if they already exist. If
`derived_only` is set, copy only derived parameters; these are real parameters
that are locked. How should we (should we at all?) generalize this to complex
parameters?

⟨*Variables: public*⟩+≡
```
      public :: var_list_init_copies
```

⟨*Variables: procedures*⟩+≡
```
      subroutine var_list_init_copies (var_list, model_vars, derived_only)
        type(var_list_t), intent(inout), target :: var_list
        type(var_list_t), intent(in) :: model_vars
        logical, intent(in), optional :: derived_only
        type(var_entry_t), pointer :: model_var, var
        type(string_t) :: name
        logical :: copy_all, locked, derived
        integer :: type
        copy_all = .true.
        if (present (derived_only))  copy_all = .not. derived_only
        model_var => model_vars%first
        do while (associated (model_var))
           name = var_entry_get_name (model_var)
           type = var_entry_get_type (model_var)
           locked = var_entry_is_locked (model_var)
```

```
                derived = type == V_REAL .and. locked
                if (copy_all .or. derived) then
                   var => var_list_get_var_ptr &
                        (var_list, name, type, follow_link = .false.)
                   if (associated (var)) then
                      call var_entry_clear_value (var)
                   else
                      allocate (var)
                      call var_entry_init_copy (var, model_var)
                      call var_list_append (var_list, var)
                   end if
                end if
                model_var => model_var%next
             end do
          end subroutine var_list_init_copies
```

Clear all previously allocated pointers to some model:

⟨*Variables: procedures*⟩+≡
```
       subroutine var_list_clear_original_pointers (var_list)
         type(var_list_t), intent(inout) :: var_list
         type(var_entry_t), pointer :: var
         var => var_list%first
         do while (associated (var))
            call var_entry_clear_original_pointer (var)
            var => var%next
         end do
       end subroutine var_list_clear_original_pointers
```

Assign the pointer to the original for a single variable.

⟨*Variables: public*⟩+≡
```
       public :: var_list_set_original_pointer
```

⟨*Variables: procedures*⟩+≡
```
       subroutine var_list_set_original_pointer (var_list, name, model_vars)
         type(var_list_t), intent(inout) :: var_list
         type(string_t), intent(in) :: name
         type(var_list_t), intent(in), target :: model_vars
         type(var_entry_t), pointer :: var, model_var
         integer :: type
         model_var => var_list_get_var_ptr (model_vars, name)
         if (associated (model_var)) then
            type = var_entry_get_type (model_var)
            var => var_list_get_var_ptr (var_list, name, type, follow_link=.false.)
            if (associated (var)) then
               call var_entry_set_original_pointer (var, model_var)
            end if
         end if
       end subroutine var_list_set_original_pointer
```

Assign pointers to the originals for all variables in the model variable list.

⟨*Variables: public*⟩+≡
```
       public :: var_list_set_original_pointers
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_set_original_pointers (var_list, model_vars)
      type(var_list_t), intent(inout) :: var_list
      type(var_list_t), intent(in), target :: model_vars
      type(var_entry_t), pointer :: var, model_var
      type(string_t) :: name
      integer :: type
      model_var => model_vars%first
      do while (associated (model_var))
         name = var_entry_get_name (model_var)
         type = var_entry_get_type (model_var)
         var => var_list_get_var_ptr (var_list, name, type, follow_link=.false.)
         if (associated (var)) then
            call var_entry_set_original_pointer (var, model_var)
         end if
         model_var => model_var%next
      end do
    end subroutine var_list_set_original_pointers
```

Synchronize the local variable list with the model variable list (which is accessed by the pointers assigned in the previous subroutine).

If `reset_pointers` is unset, do not reset pointers but just update values where a variable is a copy. Resetting pointers is done only in the local variable list.

⟨*Variables: public*⟩+≡
```
    public :: var_list_synchronize
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_synchronize (var_list, model_vars, reset_pointers)
      type(var_list_t), intent(inout) :: var_list
      type(var_list_t), intent(in), target :: model_vars
      logical, intent(in), optional :: reset_pointers
      type(var_entry_t), pointer :: var
      if (present (reset_pointers)) then
         if (reset_pointers) then
            call var_list_clear_original_pointers (var_list)
            call var_list_set_original_pointers (var_list, model_vars)
         end if
      end if
      var => var_list%first
      do while (associated (var))
         call var_entry_synchronize (var)
         var => var%next
      end do
    end subroutine var_list_synchronize
```

This is the inverse operation: synchronize the model variable list with the current variable list. This is necessary after discarding a local variable list.

For accessing all model variables, we should follow the links first, then scan the current list.

⟨*Variables: public*⟩+≡
```
    public :: var_list_restore
```

⟨Variables: procedures⟩+≡
```
recursive subroutine var_list_restore (var_list, follow_link)
  type(var_list_t), intent(inout) :: var_list
  type(var_entry_t), pointer :: var
  logical, intent(in), optional :: follow_link
  logical :: rec
  rec = .true.;  if (present (follow_link))  rec = follow_link
  if (rec .and. associated (var_list%next)) then
     call var_list_restore (var_list%next, rec)
  end if
  var => var_list%first
  do while (associated (var))
     call var_entry_restore (var)
     var => var%next
  end do
end subroutine var_list_restore
```

Mark all entries in the current variable list as undefined. This is done when a local variable list is discarded. If the local list is used again (by a loop), the entries will be re-initialized.

⟨Variables: public⟩+≡
```
public :: var_list_undefine
```

⟨Variables: procedures⟩+≡
```
recursive subroutine var_list_undefine (var_list, follow_link)
  type(var_list_t), intent(inout) :: var_list
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var
  logical :: rec
  rec = .true.;  if (present (follow_link))  rec = follow_link
  var => var_list%first
  do while (associated (var))
     call var_entry_undefine (var)
     var => var%next
  end do
  if (rec .and. associated (var_list%next)) then
     call var_list_undefine (var_list%next, follow_link=follow_link)
  end if
end subroutine var_list_undefine
```

Make a deep copy of a variable list. The copy does not contain any pointer variables. Clear the original pointer after use, since the original may be lost when the copy is in use.

⟨Variables: public⟩+≡
```
public :: var_list_init_snapshot
```

⟨Variables: procedures⟩+≡
```
recursive subroutine var_list_init_snapshot (var_list, vars_in, follow_link)
  type(var_list_t), intent(out) :: var_list
  type(var_list_t), intent(in) :: vars_in
  logical, intent(in), optional :: follow_link
  type(var_entry_t), pointer :: var, var_in
  type(var_list_t), pointer :: var_list_next
```

```
      logical :: rec
      rec = .true.;  if (present (follow_link))  rec = follow_link
      var_in => vars_in%first
      do while (associated (var_in))
         allocate (var)
         call var_entry_init_copy (var, var_in)
         call var_entry_set_original_pointer (var, var_in)
         call var_entry_synchronize (var)
         call var_entry_clear_original_pointer (var)
         call var_list_append (var_list, var)
         var_in => var_in%next
      end do
      if (rec .and. associated (vars_in%next)) then
         allocate (var_list_next)
         call var_list_init_snapshot (var_list_next, vars_in%next)
         call var_list_link (var_list, var_list_next)
      end if
   end subroutine var_list_init_snapshot
```

Check if a user variable can be set. The `new` flag is set if the user variable has
an explicit declaration. If an error occurs, return `V_NONE` as variable type.

Also determine the actual type of generic numerical variables, which enter
the procedure with type `V_NONE`.

⟨*Variables: public*⟩+≡
```
    public :: var_list_check_user_var
```

⟨*Variables: procedures*⟩+≡
```
    subroutine var_list_check_user_var (var_list, name, type, new)
      type(var_list_t), intent(in), target :: var_list
      type(string_t), intent(in) :: name
      integer, intent(inout) :: type
      logical, intent(in) :: new
      type(var_entry_t), pointer :: var
      if (string_is_observable_id (name)) then
         call msg_error ("Variable name '" // char (name) &
              // "' is reserved for an observable")
         type = V_NONE
         return
      end if
      if (string_is_integer_result_var (name))  type = V_INT
      var => var_list_get_var_ptr (var_list, name)
      if (associated (var)) then
         if (type == V_NONE) then
            type = var_entry_get_type (var)
         end if
         if (var_entry_is_locked (var)) then
            call msg_error ("Variable '" // char (name) &
                 // "' is not user-definable")
            type = V_NONE
            return
         else if (new) then
            if (var_entry_is_intrinsic (var)) then
               call msg_error ("Intrinsic variable '" &
                    // char (name) // "' redeclared")
```

```
                    type = V_NONE
                    return
                end if
                if (var_entry_get_type (var) /= type) then
                    call msg_error ("Variable '" // char (name) // "' " &
                        // "redeclared with different type")
                    type = V_NONE
                    return
                end if
            end if
        else
            if (string_is_result_var (name)) then
                call msg_error ("Result variable '" // char (name) // "' " &
                    // "set without prior integration")
                type = V_NONE
                return
            else if (string_is_num_id (name)) then
                call msg_error ("Numeric process ID '" // char (name) // "' " &
                    // "set without process declaration")
                type = V_NONE
                return
            else if (.not. new) then
                call msg_error ("Variable '" // char (name) // "' " &
                    // "set without declaration")
                type = V_NONE
                return
            end if
        end if
    end subroutine var_list_check_user_var
```

Check if a variable name is a result variable of integer type:

⟨*Variables: procedures*⟩+≡

```
    function string_is_integer_result_var (string) result (flag)
      logical :: flag
      type(string_t), intent(in) :: string
      type(string_t) :: buffer, name, separator
      buffer = string
      call split (buffer, name, "(", separator=separator)  ! ")"
      if (separator == "(") then
         select case (char (name))
         case ("num_id", "n_calls")
            flag = .true.
         case default
            flag = .false.
         end select
      else
         flag = .false.
      end if
    end function string_is_integer_result_var
```

Check if a variable name is an integration-result variable:

⟨*Variables: procedures*⟩+≡

```
    function string_is_result_var (string) result (flag)
```

```
   logical :: flag
   type(string_t), intent(in) :: string
   type(string_t) :: buffer, name, separator
   buffer = string
   call split (buffer, name, "(", separator=separator)  ! ")"
   if (separator == "(") then
      select case (char (name))
      case ("integral", "error")
         flag = .true.
      case default
         flag = .false.
      end select
   else
      flag = .false.
   end if
end function string_is_result_var
```

Check if a variable name is a numeric process ID:

⟨*Variables: procedures*⟩+≡
```
function string_is_num_id (string) result (flag)
   logical :: flag
   type(string_t), intent(in) :: string
   type(string_t) :: buffer, name, separator
   buffer = string
   call split (buffer, name, "(", separator=separator)  ! ")"
   if (separator == "(") then
      select case (char (name))
      case ("num_id")
         flag = .true.
      case default
         flag = .false.
      end select
   else
      flag = .false.
   end if
end function string_is_num_id
```

5.7   Expressions

In this module we define the structures needed to parse a user-defined expression, to compile it into an evaluation tree, and to evaluate it.

We have two flavors of expressions: one with particles and one without particles. The latter version is used for defining cut/selection criteria and for online analysis.

⟨**expressions.f90**⟩≡
  ⟨*File header*⟩

  module expressions

    use iso_c_binding !NODEP!
  ⟨*Use kinds*⟩

⟨*Use strings*⟩
```
use constants !NODEP!
```
⟨*Use file utils*⟩
```
use limits, only: FMT_19 !NODEP!
use diagnostics !NODEP!
use lorentz !NODEP!
use unit_tests
use md5
use formats
use sorting
use ifiles
use lexers
use syntax_rules
use parser
use analysis
use jets
use pdg_arrays
use subevents
use user_code_interface
use variables
```

⟨*Standard module head*⟩

⟨*Expressions: public*⟩

⟨*Expressions: types*⟩

⟨*Expressions: interfaces*⟩

⟨*Expressions: variables*⟩

```
contains
```

⟨*Expressions: procedures*⟩

⟨*Expressions: tests*⟩

```
end module expressions
```

5.7.1   Tree nodes

The evaluation tree consists of branch nodes (unary and binary) and of leaf nodes, originating from a common root. The node object should be polymorphic. For the time being, polymorphism is emulated here. This means that we have to maintain all possibilities that the node may hold, including associated procedures as pointers.

The following parameter values characterize the node. Unary and binary operators have sub-nodes. The other are leaf nodes. Possible leafs are literal constants or named-parameter references.

⟨*Expressions: types*⟩≡
```
integer, parameter :: EN_UNKNOWN = 0, EN_UNARY = 1, EN_BINARY = 2
integer, parameter :: EN_CONSTANT = 3, EN_VARIABLE = 4
integer, parameter :: EN_CONDITIONAL = 5, EN_BLOCK = 6
```

```
      integer, parameter :: EN_RECORD_CMD = 7
      integer, parameter :: EN_OBS1_INT = 11, EN_OBS2_INT = 12
      integer, parameter :: EN_OBS1_REAL = 21, EN_OBS2_REAL = 22
      integer, parameter :: EN_UOBS1_INT = 31, EN_UOBS2_INT = 32
      integer, parameter :: EN_UOBS1_REAL = 41, EN_UOBS2_REAL = 42
      integer, parameter :: EN_PRT_FUN_UNARY = 101, EN_PRT_FUN_BINARY = 102
      integer, parameter :: EN_EVAL_FUN_UNARY = 111, EN_EVAL_FUN_BINARY = 112
      integer, parameter :: EN_LOG_FUN_UNARY = 121, EN_LOG_FUN_BINARY = 122
      integer, parameter :: EN_INT_FUN_UNARY = 131, EN_INT_FUN_BINARY = 132
      integer, parameter :: EN_REAL_FUN_UNARY = 141, EN_REAL_FUN_BINARY = 142
      integer, parameter :: EN_FORMAT_STR = 161
```

⟨*Expressions: types*⟩+≡

```
    type :: eval_node_t
       private
       type(string_t) :: tag
       integer :: type = EN_UNKNOWN
       integer :: result_type = V_NONE
       type(var_list_t), pointer :: var_list => null ()
       type(string_t) :: var_name
       logical, pointer :: value_is_known => null ()
       logical,          pointer :: lval => null ()
       integer,          pointer :: ival => null ()
       real(default),    pointer :: rval => null ()
       complex(default), pointer :: cval => null ()
       type(subevt_t),   pointer :: pval => null ()
       type(pdg_array_t), pointer :: aval => null ()
       type(string_t),    pointer :: sval => null ()
       type(eval_node_t), pointer :: arg0 => null ()
       type(eval_node_t), pointer :: arg1 => null ()
       type(eval_node_t), pointer :: arg2 => null ()
       type(eval_node_t), pointer :: arg3 => null ()
       type(eval_node_t), pointer :: arg4 => null ()
       procedure(obs_unary_int),   nopass, pointer :: obs1_int  => null ()
       procedure(obs_unary_real),  nopass, pointer :: obs1_real => null ()
       procedure(obs_binary_int),  nopass, pointer :: obs2_int  => null ()
       procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
       integer, pointer :: prt_type => null ()
       integer, pointer :: index => null ()
       real(default), pointer :: tolerance => null ()
       integer, pointer :: jet_algorithm => null ()
       real(default), pointer :: jet_r => null ()
       type(prt_t), pointer :: prt1 => null ()
       type(prt_t), pointer :: prt2 => null ()
       procedure(unary_log),   nopass, pointer :: op1_log   => null ()
       procedure(unary_int),   nopass, pointer :: op1_int   => null ()
       procedure(unary_real),  nopass, pointer :: op1_real  => null ()
       procedure(unary_cmplx), nopass, pointer :: op1_cmplx => null ()
       procedure(unary_pdg),   nopass, pointer :: op1_pdg   => null ()
       procedure(unary_sev),   nopass, pointer :: op1_sev   => null ()
       procedure(unary_str),   nopass, pointer :: op1_str   => null ()
       procedure(unary_cut),   nopass, pointer :: op1_cut   => null ()
       procedure(unary_evi),   nopass, pointer :: op1_evi   => null ()
       procedure(unary_evr),   nopass, pointer :: op1_evr   => null ()
```

```
      procedure(binary_log),   nopass, pointer :: op2_log  => null ()
      procedure(binary_int),   nopass, pointer :: op2_int  => null ()
      procedure(binary_real),  nopass, pointer :: op2_real => null ()
      procedure(binary_cmplx), nopass, pointer :: op2_cmplx => null ()
      procedure(binary_pdg),   nopass, pointer :: op2_pdg  => null ()
      procedure(binary_sev),   nopass, pointer :: op2_sev  => null ()
      procedure(binary_str),   nopass, pointer :: op2_str  => null ()
      procedure(binary_cut),   nopass, pointer :: op2_cut  => null ()
      procedure(binary_evi),   nopass, pointer :: op2_evi  => null ()
      procedure(binary_evr),   nopass, pointer :: op2_evr  => null ()
    end type eval_node_t
```

Finalize a node recursively. Allocated constants are deleted, pointers are ignored.

⟨*Expressions: procedures*⟩≡
```
    recursive subroutine eval_node_final_rec (node)
      type(eval_node_t), intent(inout) :: node
      select case (node%type)
      case (EN_UNARY)
         call eval_node_final_rec (node%arg1)
      case (EN_BINARY)
         call eval_node_final_rec (node%arg1)
         call eval_node_final_rec (node%arg2)
      case (EN_CONDITIONAL)
         call eval_node_final_rec (node%arg0)
         call eval_node_final_rec (node%arg1)
         call eval_node_final_rec (node%arg2)
      case (EN_BLOCK)
         call eval_node_final_rec (node%arg0)
         call eval_node_final_rec (node%arg1)
      case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
            EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY, EN_REAL_FUN_UNARY)
         if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
         call eval_node_final_rec (node%arg1)
         deallocate (node%index)
         deallocate (node%prt1)
      case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
            EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY, EN_REAL_FUN_BINARY)
         if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
         call eval_node_final_rec (node%arg1)
         call eval_node_final_rec (node%arg2)
         deallocate (node%index)
         deallocate (node%prt1)
         deallocate (node%prt2)
      case (EN_FORMAT_STR)
         if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
         if (associated (node%arg1))  call eval_node_final_rec (node%arg1)
         deallocate (node%ival)
      case (EN_RECORD_CMD)
         if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
         if (associated (node%arg1))  call eval_node_final_rec (node%arg1)
         if (associated (node%arg2))  call eval_node_final_rec (node%arg2)
         if (associated (node%arg3))  call eval_node_final_rec (node%arg3)
```

```
          if (associated (node%arg4))  call eval_node_final_rec (node%arg4)
        end select
        select case (node%type)
        case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, EN_CONSTANT, EN_BLOCK, &
              EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
              EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
              EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
              EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
              EN_REAL_FUN_UNARY, EN_REAL_FUN_BINARY, &
              EN_FORMAT_STR, EN_RECORD_CMD)
          select case (node%result_type)
          case (V_LOG);   deallocate (node%lval)
          case (V_INT);   deallocate (node%ival)
          case (V_REAL);  deallocate (node%rval)
          case (V_CMPLX); deallocate (node%cval)
          case (V_SEV);   deallocate (node%pval)
          case (V_PDG);   deallocate (node%aval)
          case (V_STR);   deallocate (node%sval)
          end select
          deallocate (node%value_is_known)
        end select
      end subroutine eval_node_final_rec
```

Leaf nodes

Initialize a leaf node with a literal constant.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_log (node, lval)
      type(eval_node_t), intent(out) :: node
      logical, intent(in) :: lval
      node%type = EN_CONSTANT
      node%result_type = V_LOG
      allocate (node%lval, node%value_is_known)
      node%lval = lval
      node%value_is_known = .true.
    end subroutine eval_node_init_log

    subroutine eval_node_init_int (node, ival)
      type(eval_node_t), intent(out) :: node
      integer, intent(in) :: ival
      node%type = EN_CONSTANT
      node%result_type = V_INT
      allocate (node%ival, node%value_is_known)
      node%ival = ival
      node%value_is_known = .true.
    end subroutine eval_node_init_int

    subroutine eval_node_init_real (node, rval)
      type(eval_node_t), intent(out) :: node
      real(default), intent(in) :: rval
      node%type = EN_CONSTANT
      node%result_type = V_REAL
      allocate (node%rval, node%value_is_known)
```

```
      node%rval = rval
      node%value_is_known = .true.
    end subroutine eval_node_init_real

    subroutine eval_node_init_cmplx (node, cval)
      type(eval_node_t), intent(out) :: node
      complex(default), intent(in) :: cval
      node%type = EN_CONSTANT
      node%result_type = V_CMPLX
      allocate (node%cval, node%value_is_known)
      node%cval = cval
      node%value_is_known = .true.
    end subroutine eval_node_init_cmplx

    subroutine eval_node_init_subevt (node, pval)
      type(eval_node_t), intent(out) :: node
      type(subevt_t), intent(in) :: pval
      node%type = EN_CONSTANT
      node%result_type = V_SEV
      allocate (node%pval, node%value_is_known)
      node%pval = pval
      node%value_is_known = .true.
    end subroutine eval_node_init_subevt

    subroutine eval_node_init_pdg_array (node, aval)
      type(eval_node_t), intent(out) :: node
      type(pdg_array_t), intent(in) :: aval
      node%type = EN_CONSTANT
      node%result_type = V_PDG
      allocate (node%aval, node%value_is_known)
      node%aval = aval
      node%value_is_known = .true.
    end subroutine eval_node_init_pdg_array

    subroutine eval_node_init_string (node, sval)
      type(eval_node_t), intent(out) :: node
      type(string_t), intent(in) :: sval
      node%type = EN_CONSTANT
      node%result_type = V_STR
      allocate (node%sval, node%value_is_known)
      node%sval = sval
      node%value_is_known = .true.
    end subroutine eval_node_init_string
```

Initialize a leaf node with a pointer to a named parameter
⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_log_ptr (node, name, lval, is_known)
      type(eval_node_t), intent(out) :: node
      type(string_t), intent(in) :: name
      logical, intent(in), target :: lval
      logical, intent(in), target :: is_known
      node%type = EN_VARIABLE
      node%tag = name
      node%result_type = V_LOG
```

```
  node%lval => lval
  node%value_is_known => is_known
end subroutine eval_node_init_log_ptr

subroutine eval_node_init_int_ptr (node, name, ival, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  integer, intent(in), target :: ival
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_INT
  node%ival => ival
  node%value_is_known => is_known
end subroutine eval_node_init_int_ptr

subroutine eval_node_init_real_ptr (node, name, rval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  real(default), intent(in), target :: rval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_REAL
  node%rval => rval
  node%value_is_known => is_known
end subroutine eval_node_init_real_ptr

subroutine eval_node_init_cmplx_ptr (node, name, cval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  complex(default), intent(in), target :: cval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_CMPLX
  node%cval => cval
  node%value_is_known => is_known
end subroutine eval_node_init_cmplx_ptr

subroutine eval_node_init_subevt_ptr (node, name, pval, is_known)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(subevt_t), intent(in), target :: pval
  logical, intent(in), target :: is_known
  node%type = EN_VARIABLE
  node%tag = name
  node%result_type = V_SEV
  node%pval => pval
  node%value_is_known => is_known
end subroutine eval_node_init_subevt_ptr

subroutine eval_node_init_pdg_array_ptr (node, name, aval, is_known)
  type(eval_node_t), intent(out) :: node
```

```
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_PDG
    node%aval => aval
    node%value_is_known => is_known
  end subroutine eval_node_init_pdg_array_ptr

  subroutine eval_node_init_string_ptr (node, name, sval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    type(string_t), intent(in), target :: sval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_STR
    node%sval => sval
    node%value_is_known => is_known
  end subroutine eval_node_init_string_ptr
```

Branch nodes

Initialize a branch node, sub-nodes are given.

⟨*Expressions: procedures*⟩+≡
```
  subroutine eval_node_init_branch (node, tag, result_type, arg1, arg2)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: tag
    integer, intent(in) :: result_type
    type(eval_node_t), intent(in), target :: arg1
    type(eval_node_t), intent(in), target, optional :: arg2
    if (present (arg2)) then
       node%type = EN_BINARY
    else
       node%type = EN_UNARY
    end if
    node%tag = tag
    node%result_type = result_type
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    if (present (arg2))  node%arg2 => arg2
  end subroutine eval_node_init_branch
```

Allocate the node value according to the result type.

⟨*Expressions: procedures*⟩+≡
```
  subroutine eval_node_allocate_value (node)
    type(eval_node_t), intent(inout) :: node
    select case (node%result_type)
    case (V_LOG);  allocate (node%lval)
    case (V_INT);  allocate (node%ival)
    case (V_REAL); allocate (node%rval)
```

```
      case (V_CMPLX); allocate (node%cval)
      case (V_PDG);  allocate (node%aval)
      case (V_SEV);  allocate (node%pval)
         call subevt_init (node%pval)
      case (V_STR);  allocate (node%sval)
      end select
      allocate (node%value_is_known)
      node%value_is_known = .false.
    end subroutine eval_node_allocate_value
```

Initialize a node with an observable. This includes a procedure pointer. The input is a variable entry from the stack which holds the necessary information.

The user-observable version does not contain a pointer. The appropriate procedure is called directly at evaluation time. Furthermore, this is a branch node with a reference to the string-expression node which determines the name of the user procedure.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_init_obs (node, var, arg)
      type(eval_node_t), intent(out) :: node
      type(var_entry_t), intent(in), target :: var
      type(eval_node_t), intent(in), target, optional :: arg
      integer :: var_type
      node%tag = var_entry_get_name (var)
      var_type = var_entry_get_type (var)
      select case (var_type)
      case (V_OBS1_INT)
         node%type = EN_OBS1_INT
         call var_entry_assign_obs1_int_ptr (node%obs1_int, var)
      case (V_OBS2_INT)
         node%type = EN_OBS2_INT
         call var_entry_assign_obs2_int_ptr (node%obs2_int, var)
      case (V_OBS1_REAL)
         node%type = EN_OBS1_REAL
         call var_entry_assign_obs1_real_ptr (node%obs1_real, var)
      case (V_OBS2_REAL)
         node%type = EN_OBS2_REAL
         call var_entry_assign_obs2_real_ptr (node%obs2_real, var)
      case (V_UOBS1_INT)
         node%type = EN_UOBS1_INT
      case (V_UOBS2_INT)
         node%type = EN_UOBS2_INT
      case (V_UOBS1_REAL)
         node%type = EN_UOBS1_REAL
      case (V_UOBS2_REAL)
         node%type = EN_UOBS2_REAL
      end select
      select case (var_type)
      case (V_OBS1_INT, V_OBS2_INT, V_UOBS1_INT, V_UOBS2_INT)
         node%result_type = V_INT
         allocate (node%ival, node%value_is_known)
         node%value_is_known = .false.
      case (V_OBS1_REAL, V_OBS2_REAL, V_UOBS1_REAL, V_UOBS2_REAL)
         node%result_type = V_REAL
```

```
      allocate (node%rval, node%value_is_known)
      node%value_is_known = .false.
   end select
   select case (var_type)
   case (V_OBS1_INT, V_OBS1_REAL, V_UOBS1_INT, V_UOBS1_REAL)
      node%prt1 => var_entry_get_prt1_ptr (var)
   case (V_OBS2_INT, V_OBS2_REAL, V_UOBS2_INT, V_UOBS2_REAL)
      node%prt1 => var_entry_get_prt1_ptr (var)
      node%prt2 => var_entry_get_prt2_ptr (var)
   end select
   select case (var_type)
   case (V_UOBS1_INT, V_UOBS1_REAL, V_UOBS2_INT, V_UOBS2_REAL)
      if (present (arg))  node%arg0 => arg
   end select
 end subroutine eval_node_init_obs
```

Initialize a block node which contains, in addition to the expression to be eval-
uated, a variable definition. The result type is not yet assigned, because we can
compile the enclosed expression only after the var list is set up.

Note that the node always allocates a new variable list and appends it to
the current one. Thus, if the variable redefines an existing one, it only shadows
it but does not reset it. Any side-effects are therefore absent and need not be
undone outside the block.

If the flag **new** is set, a variable is (re)declared. This must not be done for
intrinsic variables. Vice versa, if the variable is not existent, the **new** flag is
required.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_init_block (node, name, type, var_def, var_list)
      type(eval_node_t), intent(out), target :: node
      type(string_t), intent(in) :: name
      integer, intent(in) :: type
      type(eval_node_t), intent(in), target :: var_def
      type(var_list_t), intent(in), target :: var_list
      node%type = EN_BLOCK
      node%tag = "var_def"
      node%var_name = name
      node%arg1 => var_def
      allocate (node%var_list)
      call var_list_link (node%var_list, var_list)
      if (var_def%type == EN_CONSTANT) then
         select case (type)
         case (V_LOG)
            call var_list_append_log  (node%var_list, name, var_def%lval)
         case (V_INT)
            call var_list_append_int  (node%var_list, name, var_def%ival)
         case (V_REAL)
            call var_list_append_real (node%var_list, name, var_def%rval)
         case (V_CMPLX)
            call var_list_append_cmplx (node%var_list, name, var_def%cval)
         case (V_PDG)
            call var_list_append_pdg_array &
                  (node%var_list, name, var_def%aval)
         case (V_SEV)
```

```
            call var_list_append_subevt &
                 (node%var_list, name, var_def%pval)
        case (V_STR)
            call var_list_append_string (node%var_list, name, var_def%sval)
        end select
    else
        select case (type)
        case (V_LOG);   call var_list_append_log_ptr &
             (node%var_list, name, var_def%lval, var_def%value_is_known)
        case (V_INT);   call var_list_append_int_ptr &
             (node%var_list, name, var_def%ival, var_def%value_is_known)
        case (V_REAL);  call var_list_append_real_ptr &
             (node%var_list, name, var_def%rval, var_def%value_is_known)
        case (V_CMPLX); call var_list_append_cmplx_ptr &
             (node%var_list, name, var_def%cval, var_def%value_is_known)
        case (V_PDG);   call var_list_append_pdg_array_ptr &
             (node%var_list, name, var_def%aval, var_def%value_is_known)
        case (V_SEV); call var_list_append_subevt_ptr &
             (node%var_list, name, var_def%pval, var_def%value_is_known)
        case (V_STR); call var_list_append_string_ptr &
             (node%var_list, name, var_def%sval, var_def%value_is_known)
        end select
    end if
end subroutine eval_node_init_block
```

Complete block initialization by assigning the expression to evaluate to `arg0`.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_set_expr (node, arg, result_type)
      type(eval_node_t), intent(inout) :: node
      type(eval_node_t), intent(in), target :: arg
      integer, intent(in), optional :: result_type
      if (present (result_type)) then
         node%result_type = result_type
      else
         node%result_type = arg%result_type
      end if
      call eval_node_allocate_value (node)
      node%arg0 => arg
    end subroutine eval_node_set_expr
```

Initialize a conditional. There are three branches: the condition (evaluates to logical) and the two alternatives (evaluate both to the same arbitrary type).

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_init_conditional (node, result_type, cond, arg1, arg2)
      type(eval_node_t), intent(out) :: node
      integer, intent(in) :: result_type
      type(eval_node_t), intent(in), target :: cond, arg1, arg2
      node%type = EN_CONDITIONAL
      node%tag = "cond"
      node%result_type = result_type
      call eval_node_allocate_value (node)
      node%arg0 => cond
      node%arg1 => arg1
```

```
      node%arg2 => arg2
   end subroutine eval_node_init_conditional
```

Initialize a recording command (which evaluates to a logical constant). The first branch is the ID of the analysis object to be filled, the optional branches 1 to 4 are the values to be recorded.

If the event-weight pointer is null, we record values with unit weight. Otherwise, we use the value pointed to as event weight.

There can be up to four arguments which represent x, y, Δy, Δx. Therefore, this is the only node type that may fill four sub-nodes.

⟨*Expressions: procedures*⟩+≡
```
   subroutine eval_node_init_record_cmd &
      (node, event_weight, id, arg1, arg2, arg3, arg4)
     type(eval_node_t), intent(out) :: node
     real(default), pointer :: event_weight
     type(eval_node_t), intent(in), target :: id
     type(eval_node_t), intent(in), optional, target :: arg1, arg2, arg3, arg4
     call eval_node_init_log (node, .true.)
     node%type = EN_RECORD_CMD
     node%rval => event_weight
     node%tag = "record_cmd"
     node%arg0 => id
     if (present (arg1)) then
        node%arg1 => arg1
        if (present (arg2)) then
           node%arg2 => arg2
           if (present (arg3)) then
              node%arg3 => arg3
              if (present (arg4)) then
                 node%arg4 => arg4
              end if
           end if
        end if
     end if
   end subroutine eval_node_init_record_cmd
```

Initialize a node for operations on subevents. The particle lists (one or two) are inserted as **arg1** and **arg2**. We allocated particle pointers as temporaries for iterating over particle lists. The procedure pointer which holds the function to evaluate for the subevents (e.g., combine, select) is also initialized.

⟨*Expressions: procedures*⟩+≡
```
   subroutine eval_node_init_prt_fun_unary (node, arg1, name, proc)
     type(eval_node_t), intent(out) :: node
     type(eval_node_t), intent(in), target :: arg1
     type(string_t), intent(in) :: name
     procedure(unary_sev) :: proc
     node%type = EN_PRT_FUN_UNARY
     node%tag = name
     node%result_type = V_SEV
     call eval_node_allocate_value (node)
     node%arg1 => arg1
     allocate (node%index, source = 0)
```

```
    allocate (node%prt1)
    node%op1_sev => proc
  end subroutine eval_node_init_prt_fun_unary

  subroutine eval_node_init_prt_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_sev) :: proc
    node%type = EN_PRT_FUN_BINARY
    node%tag = name
    node%result_type = V_SEV
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index, source = 0)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_sev => proc
  end subroutine eval_node_init_prt_fun_binary
```

Similar, but for particle-list functions that evaluate to a real value.

⟨*Expressions: procedures*⟩+≡

```
  subroutine eval_node_init_eval_fun_unary (node, arg1, name)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    node%type = EN_EVAL_FUN_UNARY
    node%tag = name
    node%result_type = V_REAL
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index, source = 0)
    allocate (node%prt1)
  end subroutine eval_node_init_eval_fun_unary

  subroutine eval_node_init_eval_fun_binary (node, arg1, arg2, name)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    node%type = EN_EVAL_FUN_BINARY
    node%tag = name
    node%result_type = V_REAL
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index, source = 0)
    allocate (node%prt1)
    allocate (node%prt2)
  end subroutine eval_node_init_eval_fun_binary
```

These are for particle-list functions that evaluate to a logical value.

⟨*Expressions: procedures*⟩+≡

```
subroutine eval_node_init_log_fun_unary (node, arg1, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1
  type(string_t), intent(in) :: name
  procedure(unary_cut) :: proc
  node%type = EN_LOG_FUN_UNARY
  node%tag = name
  node%result_type = V_LOG
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  allocate (node%index, source = 0)
  allocate (node%prt1)
  node%op1_cut => proc
end subroutine eval_node_init_log_fun_unary

subroutine eval_node_init_log_fun_binary (node, arg1, arg2, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
  procedure(binary_cut) :: proc
  node%type = EN_LOG_FUN_BINARY
  node%tag = name
  node%result_type = V_LOG
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  node%arg2 => arg2
  allocate (node%index, source = 0)
  allocate (node%prt1)
  allocate (node%prt2)
  node%op2_cut => proc
end subroutine eval_node_init_log_fun_binary
```

These are for particle-list functions that evaluate to an integer value.

⟨*Expressions: procedures*⟩+≡

```
subroutine eval_node_init_int_fun_unary (node, arg1, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1
  type(string_t), intent(in) :: name
  procedure(unary_evi) :: proc
  node%type = EN_INT_FUN_UNARY
  node%tag = name
  node%result_type = V_INT
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  allocate (node%index, source = 0)
  allocate (node%prt1)
  node%op1_evi => proc
end subroutine eval_node_init_int_fun_unary

subroutine eval_node_init_int_fun_binary (node, arg1, arg2, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
  procedure(binary_evi) :: proc
```

```
    node%type = EN_INT_FUN_BINARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index, source = 0)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_evi => proc
  end subroutine eval_node_init_int_fun_binary
```

These are for particle-list functions that evaluate to a real value.

⟨*Expressions: procedures*⟩+≡
```
  subroutine eval_node_init_real_fun_unary (node, arg1, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    procedure(unary_evr) :: proc
    node%type = EN_REAL_FUN_UNARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index, source = 0)
    allocate (node%prt1)
    node%op1_evr => proc
  end subroutine eval_node_init_real_fun_unary

  subroutine eval_node_init_real_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_evr) :: proc
    node%type = EN_REAL_FUN_BINARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index, source = 0)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_evr => proc
  end subroutine eval_node_init_real_fun_binary
```

Initialize a node for a string formatting function (sprintf).

⟨*Expressions: procedures*⟩+≡
```
  subroutine eval_node_init_format_string (node, fmt, arg, name, n_args)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), pointer :: fmt, arg
    type(string_t), intent(in) :: name
    integer, intent(in) :: n_args
```

```
      node%type = EN_FORMAT_STR
      node%tag = name
      node%result_type = V_STR
      call eval_node_allocate_value (node)
      node%arg0 => fmt
      node%arg1 => arg
      allocate (node%ival)
      node%ival = n_args
    end subroutine eval_node_init_format_string
```

If particle functions depend upon a condition (or an expression is evaluated),
the observables that can be evaluated for the given particles have to be thrown
on the local variable stack. This is done here. Each observable is initialized
with the particle pointers which have been allocated for the node.

The integer variable that is referred to by the Index pseudo-observable is
always known when it is referred to.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_set_observables (node, var_list)
      type(eval_node_t), intent(inout) :: node
      type(var_list_t), intent(in), target :: var_list
      logical, save, target :: known = .true.
      allocate (node%var_list)
      call var_list_link (node%var_list, var_list)
      allocate (node%index, source = 0)
      call var_list_append_int_ptr &
           (node%var_list, var_str ("Index"), node%index, known, intrinsic=.true.)
      if (.not. associated (node%prt2)) then
         call var_list_set_observables_unary &
              (node%var_list, node%prt1)
      else
         call var_list_set_observables_binary &
              (node%var_list, node%prt1, node%prt2)
      end if
    end subroutine eval_node_set_observables
```

Output

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_write (node, unit, indent)
      type(eval_node_t), intent(in) :: node
      integer, intent(in), optional :: unit
      integer, intent(in), optional :: indent
      integer :: u, ind
      u = output_unit (unit);  if (u < 0)  return
      ind = 0;  if (present (indent)) ind = indent
      write (u, "(A)", advance="no")  repeat ("|  ", ind) // "o "
      select case (node%type)
      case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, &
            EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
            EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
            EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
            EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
```

```
        EN_REAL_FUN_UNARY, EN_REAL_FUN_BINARY)
   write (u, "(A)", advance="no")  "[" // char (node%tag) // "] ="
case (EN_CONSTANT)
   write (u, "(A)", advance="no")  "[const] ="
case (EN_VARIABLE)
   write (u, "(A)", advance="no")  char (node%tag) // " =>"
case (EN_OBS1_INT, EN_OBS2_INT, EN_OBS1_REAL, EN_OBS2_REAL, &
     EN_UOBS1_INT, EN_UOBS2_INT, EN_UOBS1_REAL, EN_UOBS2_REAL)
   write (u, "(A)", advance="no")  char (node%tag) // " ="
case (EN_BLOCK)
   write (u, "(A)", advance="no")  "[" // char (node%tag) // "]" // &
        char (node%var_name) // " [expr] = "
case default
   write (u, "(A)", advance="no")  "[???] ="
end select
select case (node%result_type)
case (V_LOG)
   if (node%value_is_known) then
      if (node%lval) then
         write (u, "(1x,A)") "true"
      else
         write (u, "(1x,A)") "false"
      end if
   else
      write (u, "(1x,A)") "[unknown logical]"
   end if
case (V_INT)
   if (node%value_is_known) then
      write (u, "(1x,I0)")  node%ival
   else
      write (u, "(1x,A)") "[unknown integer]"
   end if
case (V_REAL)
   if (node%value_is_known) then
      write (u, "(1x," // FMT_19 // ")") node%rval
   else
      write (u, "(1x,A)") "[unknown real]"
   end if
case (V_CMPLX)
   if (node%value_is_known) then
      write (u, "(1x,'(',"// FMT_19 // ",',',"// &
           FMT_19 // ",')')") node%cval
   else
      write (u, "(1x,A)") "[unknown complex]"
   end if
case (V_SEV)
   if (char (node%tag) == "@evt") then
      write (u, "(1x,A)") "[event subevent]"
   else if (node%value_is_known) then
      call subevt_write &
           (node%pval, unit, prefix = repeat ("|  ", ind + 1))
   else
      write (u, "(1x,A)") "[unknown subevent]"
   end if
```

```
      case (V_PDG)
         write (u, "(1x)", advance="no")
         call pdg_array_write (node%aval, u);  write (u, *)
      case (V_STR)
         if (node%value_is_known) then
            write (u, "(A)")  '"' // char (node%sval) // '"'
         else
            write (u, "(1x,A)") "[unknown string]"
         end if
      case default
         write (u, "(1x,A)") "[empty]"
      end select
      select case (node%type)
      case (EN_OBS1_INT, EN_OBS1_REAL, EN_UOBS1_INT, EN_UOBS1_REAL)
         write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt1 ="
         call prt_write (node%prt1, unit)
      case (EN_OBS2_INT, EN_OBS2_REAL, EN_UOBS2_INT, EN_UOBS2_REAL)
         write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt1 ="
         call prt_write (node%prt1, unit)
         write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt2 ="
         call prt_write (node%prt2, unit)
      end select
    end subroutine eval_node_write

    recursive subroutine eval_node_write_rec (node, unit, indent)
      type(eval_node_t), intent(in) :: node
      integer, intent(in), optional :: unit
      integer, intent(in), optional :: indent
      integer :: u, ind
      u = output_unit (unit);  if (u < 0)  return
      ind = 0;  if (present (indent))  ind = indent
      call eval_node_write (node, unit, indent)
      select case (node%type)
      case (EN_UNARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
      case (EN_BINARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg2, unit, ind+1)
      case (EN_BLOCK)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg0, unit, ind+1)
      case (EN_CONDITIONAL)
         call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg2, unit, ind+1)
      case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
            EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY, EN_REAL_FUN_UNARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
```

```
      case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
            EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY, EN_REAL_FUN_BINARY)
         if (associated (node%arg0)) &
              call eval_node_write_rec (node%arg0, unit, ind+1)
         call eval_node_write_rec (node%arg1, unit, ind+1)
         call eval_node_write_rec (node%arg2, unit, ind+1)
      case (EN_RECORD_CMD)
         if (associated (node%arg1)) then
            call eval_node_write_rec (node%arg1, unit, ind+1)
            if (associated (node%arg2)) then
               call eval_node_write_rec (node%arg2, unit, ind+1)
               if (associated (node%arg3)) then
                  call eval_node_write_rec (node%arg3, unit, ind+1)
                  if (associated (node%arg4)) then
                     call eval_node_write_rec (node%arg4, unit, ind+1)
                  end if
               end if
            end if
         end if
      end select
    end subroutine eval_node_write_rec
```

5.7.2 Operation types

For the operations associated to evaluation tree nodes, we define abstract interfaces for all cases.

Particles/subevents are transferred by-reference, to avoid unnecessary copying. Therefore, subroutines instead of functions. (Furthermore, the function version of `unary_prt` triggers an obscure bug in nagfor 5.2(649) [invalid C code].)

⟨*Expressions: interfaces*⟩≡

```
    abstract interface
       logical function unary_log (arg)
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_log
    end interface
    abstract interface
       integer function unary_int (arg)
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_int
    end interface
    abstract interface
       real(default) function unary_real (arg)
         import default
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
       end function unary_real
    end interface
    abstract interface
       complex(default) function unary_cmplx (arg)
         import default
```

```
         import eval_node_t
         type(eval_node_t), intent(in) :: arg
      end function unary_cmplx
   end interface
   abstract interface
      subroutine unary_pdg (pdg_array, arg)
         import pdg_array_t
         import eval_node_t
         type(pdg_array_t), intent(out) :: pdg_array
         type(eval_node_t), intent(in) :: arg
      end subroutine unary_pdg
   end interface
   abstract interface
      subroutine unary_sev (subevt, arg, arg0)
         import subevt_t
         import eval_node_t
         type(subevt_t), intent(inout) :: subevt
         type(eval_node_t), intent(in) :: arg
         type(eval_node_t), intent(inout), optional :: arg0
      end subroutine unary_sev
   end interface
   abstract interface
      subroutine unary_str (string, arg)
         import string_t
         import eval_node_t
         type(string_t), intent(out) :: string
         type(eval_node_t), intent(in) :: arg
      end subroutine unary_str
   end interface
   abstract interface
      logical function unary_cut (arg1, arg0)
         import eval_node_t
         type(eval_node_t), intent(in) :: arg1
         type(eval_node_t), intent(inout) :: arg0
      end function unary_cut
   end interface
   abstract interface
      subroutine unary_evi (ival, arg1, arg0)
         import eval_node_t
         integer, intent(out) :: ival
         type(eval_node_t), intent(in) :: arg1
         type(eval_node_t), intent(inout), optional :: arg0
      end subroutine unary_evi
   end interface
   abstract interface
      subroutine unary_evr (rval, arg1, arg0)
         import eval_node_t, default
         real(default), intent(out) :: rval
         type(eval_node_t), intent(in) :: arg1
         type(eval_node_t), intent(inout), optional :: arg0
      end subroutine unary_evr
   end interface
   abstract interface
      logical function binary_log (arg1, arg2)
```

```
      import eval_node_t
      type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_log
end interface
abstract interface
    integer function binary_int (arg1, arg2)
      import eval_node_t
      type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_int
end interface
abstract interface
    real(default) function binary_real (arg1, arg2)
      import default
      import eval_node_t
      type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_real
end interface
abstract interface
    complex(default) function binary_cmplx (arg1, arg2)
      import default
      import eval_node_t
      type(eval_node_t), intent(in) :: arg1, arg2
    end function binary_cmplx
end interface
abstract interface
    subroutine binary_pdg (pdg_array, arg1, arg2)
      import pdg_array_t
      import eval_node_t
      type(pdg_array_t), intent(out) :: pdg_array
      type(eval_node_t), intent(in) :: arg1, arg2
    end subroutine binary_pdg
end interface
abstract interface
    subroutine binary_sev (subevt, arg1, arg2, arg0)
      import subevt_t
      import eval_node_t
      type(subevt_t), intent(inout) :: subevt
      type(eval_node_t), intent(in) :: arg1, arg2
      type(eval_node_t), intent(inout), optional :: arg0
    end subroutine binary_sev
end interface
abstract interface
    subroutine binary_str (string, arg1, arg2)
      import string_t
      import eval_node_t
      type(string_t), intent(out) :: string
      type(eval_node_t), intent(in) :: arg1, arg2
    end subroutine binary_str
end interface
abstract interface
    logical function binary_cut (arg1, arg2, arg0)
      import eval_node_t
      type(eval_node_t), intent(in) :: arg1, arg2
      type(eval_node_t), intent(inout) :: arg0
```

```
      end function binary_cut
   end interface
   abstract interface
      subroutine binary_evi (ival, arg1, arg2, arg0)
        import eval_node_t
        integer, intent(out) :: ival
        type(eval_node_t), intent(in) :: arg1, arg2
        type(eval_node_t), intent(inout), optional :: arg0
      end subroutine binary_evi
   end interface
   abstract interface
      subroutine binary_evr (rval, arg1, arg2, arg0)
        import eval_node_t, default
        real(default), intent(out) :: rval
        type(eval_node_t), intent(in) :: arg1, arg2
        type(eval_node_t), intent(inout), optional :: arg0
      end subroutine binary_evr
   end interface
```

The following subroutines set the procedure pointer:
⟨*Expressions: procedures*⟩+≡
```
   subroutine eval_node_set_op1_log (en, op)
     type(eval_node_t), intent(inout) :: en
     procedure(unary_log) :: op
     en%op1_log => op
     end subroutine eval_node_set_op1_log

   subroutine eval_node_set_op1_int (en, op)
     type(eval_node_t), intent(inout) :: en
     procedure(unary_int) :: op
     en%op1_int => op
   end subroutine eval_node_set_op1_int

   subroutine eval_node_set_op1_real (en, op)
     type(eval_node_t), intent(inout) :: en
     procedure(unary_real) :: op
     en%op1_real => op
   end subroutine eval_node_set_op1_real

   subroutine eval_node_set_op1_cmplx (en, op)
     type(eval_node_t), intent(inout) :: en
     procedure(unary_cmplx) :: op
     en%op1_cmplx => op
   end subroutine eval_node_set_op1_cmplx

   subroutine eval_node_set_op1_pdg (en, op)
     type(eval_node_t), intent(inout) :: en
     procedure(unary_pdg) :: op
     en%op1_pdg => op
   end subroutine eval_node_set_op1_pdg

   subroutine eval_node_set_op1_sev (en, op)
     type(eval_node_t), intent(inout) :: en
     procedure(unary_sev) :: op
```

```
    en%op1_sev => op
end subroutine eval_node_set_op1_sev

subroutine eval_node_set_op1_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_str) :: op
  en%op1_str => op
end subroutine eval_node_set_op1_str

subroutine eval_node_set_op2_log (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_log) :: op
  en%op2_log => op
end subroutine eval_node_set_op2_log

subroutine eval_node_set_op2_int (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_int) :: op
  en%op2_int => op
end subroutine eval_node_set_op2_int

subroutine eval_node_set_op2_real (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_real) :: op
  en%op2_real => op
end subroutine eval_node_set_op2_real

subroutine eval_node_set_op2_cmplx (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_cmplx) :: op
  en%op2_cmplx => op
end subroutine eval_node_set_op2_cmplx

subroutine eval_node_set_op2_pdg (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_pdg) :: op
  en%op2_pdg => op
end subroutine eval_node_set_op2_pdg

subroutine eval_node_set_op2_sev (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_sev) :: op
  en%op2_sev => op
end subroutine eval_node_set_op2_sev

subroutine eval_node_set_op2_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_str) :: op
  en%op2_str => op
end subroutine eval_node_set_op2_str
```

5.7.3   Specific operators

Our expression syntax contains all Fortran functions that make sense. These functions have to be provided in a form that they can be used in procedures pointers, and have the abstract interfaces above. For some intrinsic functions, we could use specific versions provided by Fortran directly. However, this has two drawbacks: (i) We should work with the values instead of the eval-nodes as argument, which complicates the interface; (ii) more importantly, the `default` real type need not be equivalent to double precision. This would, at least, introduce system dependencies. Finally, for operators there are no specific versions.

Therefore, we write wrappers for all possible functions, at the expense of some overhead.

Binary numerical functions

⟨*Expressions: procedures*⟩+≡
```
    integer function add_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival + en2%ival
    end function add_ii
    real(default) function add_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival + en2%rval
    end function add_ir
    complex(default) function add_ic (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival + en2%cval
    end function add_ic
    real(default) function add_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval + en2%ival
    end function add_ri
    complex(default) function add_ci (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%cval + en2%ival
    end function add_ci
    complex(default) function add_cr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%cval + en2%rval
    end function add_cr
    complex(default) function add_rc (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval + en2%cval
    end function add_rc
    real(default) function add_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval + en2%rval
    end function add_rr
    complex(default) function add_cc (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%cval + en2%cval
    end function add_cc

    integer function sub_ii (en1, en2) result (y)
```

```
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival - en2%ival
  end function sub_ii
  real(default) function sub_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival - en2%rval
  end function sub_ir
  real(default) function sub_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval - en2%ival
  end function sub_ri
  complex(default) function sub_ic (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival - en2%cval
  end function sub_ic
  complex(default) function sub_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval - en2%ival
  end function sub_ci
  complex(default) function sub_cr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval - en2%rval
  end function sub_cr
  complex(default) function sub_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval - en2%cval
  end function sub_rc
  real(default) function sub_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval - en2%rval
  end function sub_rr
  complex(default) function sub_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval - en2%cval
  end function sub_cc

  integer function mul_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival * en2%ival
  end function mul_ii
  real(default) function mul_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival * en2%rval
  end function mul_ir
  real(default) function mul_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval * en2%ival
  end function mul_ri
  complex(default) function mul_ic (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival * en2%cval
  end function mul_ic
  complex(default) function mul_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
```

```
    y = en1%cval * en2%ival
  end function mul_ci
complex(default) function mul_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval * en2%cval
  end function mul_rc
complex(default) function mul_cr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval * en2%rval
  end function mul_cr
real(default) function mul_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval * en2%rval
  end function mul_rr
complex(default) function mul_cc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval * en2%cval
  end function mul_cc

integer function div_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (en2%ival == 0) then
       if (en1%ival >= 0) then
          call msg_warning ("division by zero: " // int2char (en1%ival) // &
             " / 0 ; result set to 0")
       else
          call msg_warning ("division by zero: (" // int2char (en1%ival) // &
             ") / 0 ; result set to 0")
       end if
       y = 0
       return
    end if
    y = en1%ival / en2%ival
  end function div_ii
real(default) function div_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival / en2%rval
  end function div_ir
real(default) function div_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval / en2%ival
  end function div_ri
complex(default) function div_ic (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival / en2%cval
  end function div_ic
complex(default) function div_ci (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%cval / en2%ival
  end function div_ci
complex(default) function div_rc (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval / en2%cval
  end function div_rc
```

```
complex(default) function div_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%rval
end function div_cr
real(default) function div_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%rval
end function div_rr
complex(default) function div_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%cval
end function div_cc

integer function pow_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  integer :: a, b
  real(default) :: rres
  a = en1%ival
  b = en2%ival
  if ((a == 0) .and. (b < 0)) then
     call msg_warning ("division by zero: " // int2char (a) // &
        " ^ (" // int2char (b) // ") ; result set to 0")
     y = 0
     return
  end if
  rres = real(a, default) ** b
  y = rres
  if (real(y, default) /= rres) then
     if (b < 0) then
        call msg_warning ("result of all-integer operation " // &
              int2char (a) // " ^ (" // int2char (b) // &
              ") has been trucated to "// int2char (y), &
           (/ var_str ("Chances are that you want to use " // &
              "reals instead of integers at this point.") /))
     else
        call msg_warning ("integer overflow in " // int2char (a) // &
              " ^ " // int2char (b) // " ; result is " // int2char (y), &
           (/ var_str ("Using reals instead of integers might help.") /))
     end if
  end if
end function pow_ii
real(default) function pow_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%ival
end function pow_ri
complex(default) function pow_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%ival
end function pow_ci
real(default) function pow_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%rval
end function pow_ir
real(default) function pow_rr (en1, en2) result (y)
```

```
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%rval
end function pow_rr
complex(default) function pow_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%rval
end function pow_cr
complex(default) function pow_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%cval
end function pow_ic
complex(default) function pow_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%cval
end function pow_rc
complex(default) function pow_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%cval
end function pow_cc

integer function max_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (en1%ival, en2%ival)
end function max_ii
real(default) function max_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (real (en1%ival, default), en2%rval)
end function max_ir
real(default) function max_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (en1%rval, real (en2%ival, default))
end function max_ri
real(default) function max_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (en1%rval, en2%rval)
end function max_rr
integer function min_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (en1%ival, en2%ival)
end function min_ii
real(default) function min_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (real (en1%ival, default), en2%rval)
end function min_ir
real(default) function min_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (en1%rval, real (en2%ival, default))
end function min_ri
real(default) function min_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (en1%rval, en2%rval)
end function min_rr

integer function mod_ii (en1, en2) result (y)
```

```
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%ival, en2%ival)
  end function mod_ii
  real(default) function mod_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (real (en1%ival, default), en2%rval)
  end function mod_ir
  real(default) function mod_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%rval, real (en2%ival, default))
  end function mod_ri
  real(default) function mod_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = mod (en1%rval, en2%rval)
  end function mod_rr
  integer function modulo_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%ival, en2%ival)
  end function modulo_ii
  real(default) function modulo_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (real (en1%ival, default), en2%rval)
  end function modulo_ir
  real(default) function modulo_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%rval, real (en2%ival, default))
  end function modulo_ri
  real(default) function modulo_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = modulo (en1%rval, en2%rval)
  end function modulo_rr
```

Unary numeric functions

⟨*Expressions: procedures*⟩+≡
```
  real(default) function real_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%ival
  end function real_i
  real(default) function real_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%cval
  end function real_c
  integer function int_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%rval
  end function int_r
  complex(default) function cmplx_i (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = en%ival
  end function cmplx_i
  integer function int_c (en) result (y)
    type(eval_node_t), intent(in) :: en
```

```fortran
    y = en%cval
end function int_c
complex(default) function cmplx_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%rval
end function cmplx_r
integer function nint_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = nint (en%rval)
end function nint_r
integer function floor_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = floor (en%rval)
end function floor_r
integer function ceiling_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = ceiling (en%rval)
end function ceiling_r

integer function neg_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%ival
end function neg_i
real(default) function neg_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%rval
end function neg_r
complex(default) function neg_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%cval
end function neg_c
integer function abs_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%ival)
end function abs_i
real(default) function abs_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%rval)
end function abs_r
real(default) function abs_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%cval)
end function abs_c
integer function sgn_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sign (1, en%ival)
end function sgn_i
real(default) function sgn_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sign (1._default, en%rval)
end function sgn_r

real(default) function sqrt_r (en) result (y)
  type(eval_node_t), intent(in) :: en
```

```
    y = sqrt (en%rval)
end function sqrt_r
real(default) function exp_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = exp (en%rval)
end function exp_r
real(default) function log_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log (en%rval)
end function log_r
real(default) function log10_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log10 (en%rval)
end function log10_r

complex(default) function sqrt_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sqrt (en%cval)
end function sqrt_c
complex(default) function exp_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = exp (en%cval)
end function exp_c
complex(default) function log_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log (en%cval)
end function log_c

real(default) function sin_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sin (en%rval)
end function sin_r
real(default) function cos_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = cos (en%rval)
end function cos_r
real(default) function tan_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = tan (en%rval)
end function tan_r
real(default) function asin_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = asin (en%rval)
end function asin_r
real(default) function acos_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = acos (en%rval)
end function acos_r
real(default) function atan_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = atan (en%rval)
end function atan_r

complex(default) function sin_c (en) result (y)
```

```
        type(eval_node_t), intent(in) :: en
        y = sin (en%cval)
      end function sin_c
      complex(default) function cos_c (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = cos (en%cval)
      end function cos_c

      real(default) function sinh_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = sinh (en%rval)
      end function sinh_r
      real(default) function cosh_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = cosh (en%rval)
      end function cosh_r
      real(default) function tanh_r (en) result (y)
        type(eval_node_t), intent(in) :: en
        y = tanh (en%rval)
      end function tanh_r
    !!! These are F2008 additions but accepted by nagfor 5.3 and gfortran 4.6+
    !!! Currently not used.
    !   real(default) function asinh_r (en) result (y)
    !     type(eval_node_t), intent(in) :: en
    !     y = asinh (en%rval)
    !   end function asinh_r
    !   real(default) function acosh_r (en) result (y)
    !     type(eval_node_t), intent(in) :: en
    !     y = acosh (en%rval)
    !   end function acosh_r
    !   real(default) function atanh_r (en) result (y)
    !     type(eval_node_t), intent(in) :: en
    !     y = atanh (en%rval)
    !   end function atanh_r
```

Binary logical functions

Logical expressions:

⟨*Expressions: procedures*⟩+≡
```
      logical function ignore_first_ll (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en2%lval
      end function ignore_first_ll
      logical function or_ll (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%lval .or. en2%lval
      end function or_ll
      logical function and_ll (en1, en2) result (y)
        type(eval_node_t), intent(in) :: en1, en2
        y = en1%lval .and. en2%lval
      end function and_ll
```

Comparisons:

⟨Expressions: procedures⟩+≡

```
    logical function comp_lt_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival < en2%ival
    end function comp_lt_ii
    logical function comp_lt_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival < en2%rval
    end function comp_lt_ir
    logical function comp_lt_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval < en2%ival
    end function comp_lt_ri
    logical function comp_lt_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval < en2%rval
    end function comp_lt_rr

    logical function comp_gt_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival > en2%ival
    end function comp_gt_ii
    logical function comp_gt_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival > en2%rval
    end function comp_gt_ir
    logical function comp_gt_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval > en2%ival
    end function comp_gt_ri
    logical function comp_gt_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval > en2%rval
    end function comp_gt_rr

    logical function comp_le_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival <= en2%ival
    end function comp_le_ii
    logical function comp_le_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%ival <= en2%rval
    end function comp_le_ir
    logical function comp_le_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval <= en2%ival
    end function comp_le_ri
    logical function comp_le_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%rval <= en2%rval
    end function comp_le_rr

    logical function comp_ge_ii (en1, en2) result (y)
```

```
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival >= en2%ival
  end function comp_ge_ii
  logical function comp_ge_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival >= en2%rval
  end function comp_ge_ir
  logical function comp_ge_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval >= en2%ival
  end function comp_ge_ri
  logical function comp_ge_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval >= en2%rval
  end function comp_ge_rr

  logical function comp_eq_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival == en2%ival
  end function comp_eq_ii
  logical function comp_eq_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival == en2%rval
  end function comp_eq_ir
  logical function comp_eq_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval == en2%ival
  end function comp_eq_ri
  logical function comp_eq_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval == en2%rval
  end function comp_eq_rr
  logical function comp_eq_ss (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%sval == en2%sval
  end function comp_eq_ss

  logical function comp_ne_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival /= en2%ival
  end function comp_ne_ii
  logical function comp_ne_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%ival /= en2%rval
  end function comp_ne_ir
  logical function comp_ne_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval /= en2%ival
  end function comp_ne_ri
  logical function comp_ne_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    y = en1%rval /= en2%rval
  end function comp_ne_rr
  logical function comp_ne_ss (en1, en2) result (y)
```

```
      type(eval_node_t), intent(in) :: en1, en2
      y = en1%sval /= en2%sval
   end function comp_ne_ss
```

Comparisons with tolerance:

⟨*Expressions: procedures*⟩+≡

```
   logical function comp_se_ii (en1, en2) result (y)
     type(eval_node_t), intent(in) :: en1, en2
     if (associated (en1%tolerance)) then
        y = abs (en1%ival - en2%ival) <= en1%tolerance
     else
        y = en1%ival == en2%ival
     end if
   end function comp_se_ii
   logical function comp_se_ri (en1, en2) result (y)
     type(eval_node_t), intent(in) :: en1, en2
     if (associated (en1%tolerance)) then
        y = abs (en1%rval - en2%ival) <= en1%tolerance
     else
        y = en1%rval == en2%ival
     end if
   end function comp_se_ri
   logical function comp_se_ir (en1, en2) result (y)
     type(eval_node_t), intent(in) :: en1, en2
     if (associated (en1%tolerance)) then
        y = abs (en1%ival - en2%rval) <= en1%tolerance
     else
        y = en1%ival == en2%rval
     end if
   end function comp_se_ir
   logical function comp_se_rr (en1, en2) result (y)
     type(eval_node_t), intent(in) :: en1, en2
     if (associated (en1%tolerance)) then
        y = abs (en1%rval - en2%rval) <= en1%tolerance
     else
        y = en1%rval == en2%rval
     end if
   end function comp_se_rr
   logical function comp_ns_ii (en1, en2) result (y)
     type(eval_node_t), intent(in) :: en1, en2
     if (associated (en1%tolerance)) then
        y = abs (en1%ival - en2%ival) > en1%tolerance
     else
        y = en1%ival /= en2%ival
     end if
   end function comp_ns_ii
   logical function comp_ns_ri (en1, en2) result (y)
     type(eval_node_t), intent(in) :: en1, en2
     if (associated (en1%tolerance)) then
        y = abs (en1%rval - en2%ival) > en1%tolerance
     else
        y = en1%rval /= en2%ival
     end if
   end function comp_ns_ri
```

```fortran
logical function comp_ns_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%ival - en2%rval) > en1%tolerance
  else
     y = en1%ival /= en2%rval
  end if
end function comp_ns_ir
logical function comp_ns_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%rval - en2%rval) > en1%tolerance
  else
     y = en1%rval /= en2%rval
  end if
end function comp_ns_rr

logical function comp_ls_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival <= en2%ival + en1%tolerance
  else
     y = en1%ival <= en2%ival
  end if
end function comp_ls_ii
logical function comp_ls_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval <= en2%ival + en1%tolerance
  else
     y = en1%rval <= en2%ival
  end if
end function comp_ls_ri
logical function comp_ls_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival <= en2%rval + en1%tolerance
  else
     y = en1%ival <= en2%rval
  end if
end function comp_ls_ir
logical function comp_ls_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval <= en2%rval + en1%tolerance
  else
     y = en1%rval <= en2%rval
  end if
end function comp_ls_rr

logical function comp_ll_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival < en2%ival - en1%tolerance
```

```fortran
      else
         y = en1%ival < en2%ival
      end if
  end function comp_ll_ii
  logical function comp_ll_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%rval < en2%ival - en1%tolerance
    else
       y = en1%rval < en2%ival
    end if
  end function comp_ll_ri
  logical function comp_ll_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%ival < en2%rval - en1%tolerance
    else
       y = en1%ival < en2%rval
    end if
  end function comp_ll_ir
  logical function comp_ll_rr (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%rval < en2%rval - en1%tolerance
    else
       y = en1%rval < en2%rval
    end if
  end function comp_ll_rr

  logical function comp_gs_ii (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%ival >= en2%ival - en1%tolerance
    else
       y = en1%ival >= en2%ival
    end if
  end function comp_gs_ii
  logical function comp_gs_ri (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%rval >= en2%ival - en1%tolerance
    else
       y = en1%rval >= en2%ival
    end if
  end function comp_gs_ri
  logical function comp_gs_ir (en1, en2) result (y)
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%ival >= en2%rval - en1%tolerance
    else
       y = en1%ival >= en2%rval
    end if
  end function comp_gs_ir
  logical function comp_gs_rr (en1, en2) result (y)
```

```
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = en1%rval >= en2%rval - en1%tolerance
      else
         y = en1%rval >= en2%rval
      end if
   end function comp_gs_rr

   logical function comp_gg_ii (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = en1%ival > en2%ival + en1%tolerance
      else
         y = en1%ival > en2%ival
      end if
   end function comp_gg_ii
   logical function comp_gg_ri (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = en1%rval > en2%ival + en1%tolerance
      else
         y = en1%rval > en2%ival
      end if
   end function comp_gg_ri
   logical function comp_gg_ir (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = en1%ival > en2%rval + en1%tolerance
      else
         y = en1%ival > en2%rval
      end if
   end function comp_gg_ir
   logical function comp_gg_rr (en1, en2) result (y)
      type(eval_node_t), intent(in) :: en1, en2
      if (associated (en1%tolerance)) then
         y = en1%rval > en2%rval + en1%tolerance
      else
         y = en1%rval > en2%rval
      end if
   end function comp_gg_rr
```

Unary logical functions

⟨*Expressions: procedures*⟩+≡
```
   logical function not_l (en) result (y)
      type(eval_node_t), intent(in) :: en
      y = .not. en%lval
   end function not_l
```

Unary PDG-array functions

Make a PDG-array object from an integer.

⟨Expressions: procedures⟩+≡
```
    subroutine pdg_i (pdg_array, en)
      type(pdg_array_t), intent(out) :: pdg_array
      type(eval_node_t), intent(in) :: en
      pdg_array = en%ival
    end subroutine pdg_i
```

Binary PDG-array functions

Concatenate two PDG-array objects.

⟨Expressions: procedures⟩+≡
```
    subroutine concat_cc (pdg_array, en1, en2)
      type(pdg_array_t), intent(out) :: pdg_array
      type(eval_node_t), intent(in) :: en1, en2
      pdg_array = en1%aval // en2%aval
    end subroutine concat_cc
```

Unary particle-list functions

Combine all particles of the first argument. If en0 is present, create a mask which is true only for those particles that pass the test.

⟨Expressions: procedures⟩+≡
```
    subroutine collect_p (subevt, en1, en0)
      type(subevt_t), intent(inout) :: subevt
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:), allocatable :: mask1
      integer :: n, i
      n = subevt_get_length (en1%pval)
      allocate (mask1 (n))
      if (present (en0)) then
         do i = 1, n
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            call eval_node_evaluate (en0)
            mask1(i) = en0%lval
         end do
      else
         mask1 = .true.
      end if
      call subevt_collect (subevt, en1%pval, mask1)
    end subroutine collect_p
```

Cluster the particles of the first argument. If en0 is present, create a mask which is true only for those particles that pass the test.

⟨Expressions: procedures⟩+≡
```
    subroutine cluster_p (subevt, en1, en0)
      type(subevt_t), intent(inout) :: subevt
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout), optional :: en0
```

```
      logical, dimension(:), allocatable :: mask1
      integer :: n, i
      ! Should not be hardcoded!
      type(jet_definition_t) :: jet_def
      call jet_def%init (en1%jet_algorithm, en1%jet_r)
      n = subevt_get_length (en1%pval)
      allocate (mask1 (n))
      if (present (en0)) then
         do i = 1, n
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            call eval_node_evaluate (en0)
            mask1(i) = en0%lval
         end do
      else
         mask1 = .true.
      end if
      call subevt_cluster (subevt, en1%pval, mask1, jet_def)
      call jet_def%final ()
   end subroutine cluster_p
```

Select all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test.

⟨*Expressions: procedures*⟩+≡

```
   subroutine select_p (subevt, en1, en0)
      type(subevt_t), intent(inout) :: subevt
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:), allocatable :: mask1
      integer :: n, i
      n = subevt_get_length (en1%pval)
      allocate (mask1 (n))
      if (present (en0)) then
         do i = 1, subevt_get_length (en1%pval)
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            call eval_node_evaluate (en0)
            mask1(i) = en0%lval
         end do
      else
         mask1 = .true.
      end if
      call subevt_select (subevt, en1%pval, mask1)
   end subroutine select_p
```

Extract the particle with index given by `en0` from the argument list. Negative indices count from the end. If `en0` is absent, extract the first particle. The result is a list with a single entry, or no entries if the original list was empty or if the index is out of range.

This function has no counterpart with two arguments.

⟨*Expressions: procedures*⟩+≡

```
   subroutine extract_p (subevt, en1, en0)
      type(subevt_t), intent(inout) :: subevt
```

```
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    integer :: index
    if (present (en0)) then
       call eval_node_evaluate (en0)
       select case (en0%result_type)
       case (V_INT);  index = en0%ival
       case default
          call eval_node_write (en0)
          call msg_fatal (" Index parameter of 'extract' must be integer.")
       end select
    else
       index = 1
    end if
    call subevt_extract (subevt, en1%pval, index)
  end subroutine extract_p
```

Sort the subevent according to the result of evaluating en0. If en0 is absent, sort by default method (PDG code, particles before antiparticles).

⟨*Expressions: procedures*⟩+≡
```
  subroutine sort_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    integer, dimension(:), allocatable :: ival
    real(default), dimension(:), allocatable :: rval
    integer :: i, n
    n = subevt_get_length (en1%pval)
    if (present (en0)) then
       select case (en0%result_type)
       case (V_INT);  allocate (ival (n))
       case (V_REAL); allocate (rval (n))
       end select
       do i = 1, n
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          call eval_node_evaluate (en0)
          select case (en0%result_type)
          case (V_INT);  ival(i) = en0%ival
          case (V_REAL); rval(i) = en0%rval
          end select
       end do
       select case (en0%result_type)
       case (V_INT);  call subevt_sort (subevt, en1%pval, ival)
       case (V_REAL); call subevt_sort (subevt, en1%pval, rval)
       end select
    else
       call subevt_sort (subevt, en1%pval)
    end if
  end subroutine sort_p
```

The following functions return a logical value. all evaluates to true if the condition en0 is true for all elements of the subevent. any and no are analogous.

⟨Expressions: procedures⟩+≡

```
    function all_p (en1, en0) result (lval)
      logical :: lval
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout) :: en0
      integer :: i, n
      n = subevt_get_length (en1%pval)
      lval = .true.
      do i = 1, n
         en0%index = i
         en0%prt1 = subevt_get_prt (en1%pval, i)
         call eval_node_evaluate (en0)
         lval = en0%lval
         if (.not. lval)  exit
      end do
    end function all_p

    function any_p (en1, en0) result (lval)
      logical :: lval
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout) :: en0
      integer :: i, n
      n = subevt_get_length (en1%pval)
      lval = .false.
      do i = 1, n
         en0%index = i
         en0%prt1 = subevt_get_prt (en1%pval, i)
         call eval_node_evaluate (en0)
         lval = en0%lval
         if (lval)  exit
      end do
    end function any_p

    function no_p (en1, en0) result (lval)
      logical :: lval
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout) :: en0
      integer :: i, n
      n = subevt_get_length (en1%pval)
      lval = .true.
      do i = 1, n
         en0%index = i
         en0%prt1 = subevt_get_prt (en1%pval, i)
         call eval_node_evaluate (en0)
         lval = .not. en0%lval
         if (lval)  exit
      end do
    end function no_p
```

This is the interface to user-supplied observables. The node en0 evaluates to a
string that indicates the procedure name. We search for the procedure in the
dynamic library and load it into the procedure pointer which is then called.
en1 is the subevent on which the external code operates. The external function

returns a `c_int`, which we translate into a real value.

⟨*Expressions: procedures*⟩+≡

```
    function user_obs_int_p (en0, prt1) result (ival)
      integer :: ival
      type(eval_node_t), intent(inout) :: en0
      type(prt_t), intent(in) :: prt1
      type(string_t) :: name
      procedure(user_obs_int_unary), pointer :: user_obs
      call eval_node_evaluate (en0)
      if (en0%value_is_known) then
          select case (en0%result_type)
          case (V_STR);  name = en0%sval
          case default
             call msg_bug ("user_obs: procedure name must be a string")
             name = ""
          end select
          call c_f_procpointer (user_code_find_proc (name), user_obs)
          ival = user_obs (c_prt (prt1))
      else
          call eval_node_write_rec (en0)
          call msg_fatal ("User observable name is undefined")
      end if
    end function user_obs_int_p

    function user_obs_real_p (en0, prt1) result (rval)
      real(default) :: rval
      type(eval_node_t), intent(inout) :: en0
      type(prt_t), intent(in) :: prt1
      type(string_t) :: name
      procedure(user_obs_real_unary), pointer :: user_obs
      call eval_node_evaluate (en0)
      if (en0%value_is_known) then
          select case (en0%result_type)
          case (V_STR);  name = en0%sval
          case default
             call msg_bug ("user_obs: procedure name must be a string")
             name = ""
          end select
          call c_f_procpointer (user_code_find_proc (name), user_obs)
          rval = user_obs (c_prt (prt1))
      else
          call eval_node_write_rec (en0)
          call msg_fatal ("User observable name is undefined")
      end if
    end function user_obs_real_p
```

This is the interface to user-supplied cut code. The node **en0** evaluates to a string that indicates the procedure name.

⟨*Expressions: procedures*⟩+≡

```
    function user_cut_p (en1, en0) result (lval)
      logical :: lval
      type(eval_node_t), intent(in) :: en1
      type(eval_node_t), intent(inout) :: en0
```

```
      type(string_t) :: name
      procedure(user_cut_fun), pointer :: user_cut
      call eval_node_evaluate (en0)
      select case (en0%result_type)
      case (V_STR);  name = en0%sval
      case default
         call msg_bug ("user_cut: procedure name must be a string")
         name = ""
      end select
      call c_f_procpointer (user_code_find_proc (name), user_cut)
      lval = user_cut (c_prt (en1%pval), &
                       int (subevt_get_length (en1%pval), kind=c_int)) &
            /= 0
   end function user_cut_p
```

The following function returns an integer value, namely the number of particles for which the condition is true. If there is no condition, it returns simply the length of the subevent.

A function would be more natural. Making it a subroutine avoids another compiler bug (internal error in nagfor 5.2 (649)). (See the interface unary_evi.)
⟨*Expressions: procedures*⟩+≡
```
   subroutine count_a (ival, en1, en0)
     integer, intent(out) :: ival
     type(eval_node_t), intent(in) :: en1
     type(eval_node_t), intent(inout), optional :: en0
     integer :: i, n, count
     n = subevt_get_length (en1%pval)
     if (present (en0)) then
        count = 0
        do i = 1, n
           en0%index = i
           en0%prt1 = subevt_get_prt (en1%pval, i)
           call eval_node_evaluate (en0)
           if (en0%lval)  count = count + 1
        end do
        ival = count
     else
        ival = n
     end if
   end subroutine count_a
```

This evaluates a user-defined event-shape observable for the current subevent.
⟨*Expressions: procedures*⟩+≡
```
   subroutine user_event_shape_a (rval, en1, en0)
     real(default), intent(out) :: rval
     type(eval_node_t), intent(in) :: en1
     type(eval_node_t), intent(inout), optional :: en0
     type(string_t) :: name
     procedure(user_event_shape_fun), pointer :: user_event_shape
     if (.not. present (en0))  call msg_bug &
          ("user_event_shape called without procedure name")
     call eval_node_evaluate (en0)
     select case (en0%result_type)
```

```
      case (V_STR);  name = en0%sval
      case default
         call msg_bug ("user_event_shape: procedure name must be a string")
         name = ""
      end select
      call c_f_procpointer (user_code_find_proc (name), user_event_shape)
      rval = user_event_shape (c_prt (en1%pval), &
                               int (subevt_get_length (en1%pval), kind=c_int))
    end subroutine user_event_shape_a
```

Binary particle-list functions

This joins two subevents, stored in the evaluation nodes `en1` and `en2`. If `en0` is also present, it amounts to a logical test returning true or false for every pair of particles. A particle of the second list gets a mask entry only if it passes the test for all particles of the first list.

⟨*Expressions: procedures*⟩+≡

```
    subroutine join_pp (subevt, en1, en2, en0)
      type(subevt_t), intent(inout) :: subevt
      type(eval_node_t), intent(in) :: en1, en2
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:), allocatable :: mask2
      integer :: i, j, n1, n2
      n1 = subevt_get_length (en1%pval)
      n2 = subevt_get_length (en2%pval)
      allocate (mask2 (n2))
      mask2 = .true.
      if (present (en0)) then
         do i = 1, n1
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            do j = 1, n2
               en0%prt2 = subevt_get_prt (en2%pval, j)
               call eval_node_evaluate (en0)
               mask2(j) = mask2(j) .and. en0%lval
            end do
         end do
      end if
      call subevt_join (subevt, en1%pval, en2%pval, mask2)
    end subroutine join_pp
```

Combine two subevents, i.e., make a list of composite particles built from all possible particle pairs from the two lists. If `en0` is present, create a mask which is true only for those pairs that pass the test.

⟨*Expressions: procedures*⟩+≡

```
    subroutine combine_pp (subevt, en1, en2, en0)
      type(subevt_t), intent(inout) :: subevt
      type(eval_node_t), intent(in) :: en1, en2
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:,:), allocatable :: mask12
      integer :: i, j, n1, n2
      n1 = subevt_get_length (en1%pval)
```

```
            n2 = subevt_get_length (en2%pval)
            if (present (en0)) then
                allocate (mask12 (n1, n2))
                do i = 1, n1
                    en0%index = i
                    en0%prt1 = subevt_get_prt (en1%pval, i)
                    do j = 1, n2
                        en0%prt2 = subevt_get_prt (en2%pval, j)
                        call eval_node_evaluate (en0)
                        mask12(i,j) = en0%lval
                    end do
                end do
                call subevt_combine (subevt, en1%pval, en2%pval, mask12)
            else
                call subevt_combine (subevt, en1%pval, en2%pval)
            end if
        end subroutine combine_pp
```

Combine all particles of the first argument. If en0 is present, create a mask
which is true only for those particles that pass the test w.r.t. all particles in the
second argument. If en0 is absent, the second argument is ignored.

⟨*Expressions: procedures*⟩+≡

```
        subroutine collect_pp (subevt, en1, en2, en0)
            type(subevt_t), intent(inout) :: subevt
            type(eval_node_t), intent(in) :: en1, en2
            type(eval_node_t), intent(inout), optional :: en0
            logical, dimension(:), allocatable :: mask1
            integer :: i, j, n1, n2
            n1 = subevt_get_length (en1%pval)
            n2 = subevt_get_length (en2%pval)
            allocate (mask1 (n1))
            mask1 = .true.
            if (present (en0)) then
                do i = 1, n1
                    en0%index = i
                    en0%prt1 = subevt_get_prt (en1%pval, i)
                    do j = 1, n2
                        en0%prt2 = subevt_get_prt (en2%pval, j)
                        call eval_node_evaluate (en0)
                        mask1(i) = mask1(i) .and. en0%lval
                    end do
                end do
            end if
            call subevt_collect (subevt, en1%pval, mask1)
        end subroutine collect_pp
```

Select all particles of the first argument. If en0 is present, create a mask which
is true only for those particles that pass the test w.r.t. all particles in the
second argument. If en0 is absent, the second argument is ignored, and the first
argument is transferred unchanged. (This case is not very useful, of course.)

⟨*Expressions: procedures*⟩+≡

```
        subroutine select_pp (subevt, en1, en2, en0)
            type(subevt_t), intent(inout) :: subevt
```

```
      type(eval_node_t), intent(in) :: en1, en2
      type(eval_node_t), intent(inout), optional :: en0
      logical, dimension(:), allocatable :: mask1
      integer :: i, j, n1, n2
      n1 = subevt_get_length (en1%pval)
      n2 = subevt_get_length (en2%pval)
      allocate (mask1 (n1))
      mask1 = .true.
      if (present (en0)) then
         do i = 1, n1
            en0%index = i
            en0%prt1 = subevt_get_prt (en1%pval, i)
            do j = 1, n2
               en0%prt2 = subevt_get_prt (en2%pval, j)
               call eval_node_evaluate (en0)
               mask1(i) = mask1(i) .and. en0%lval
            end do
         end do
      end if
      call subevt_select (subevt, en1%pval, mask1)
   end subroutine select_pp
```

Sort the first subevent according to the result of evaluating en0. From the second subevent, only the first element is taken as reference. If en0 is absent, we sort by default method (PDG code, particles before antiparticles).

⟨*Expressions: procedures*⟩+≡

```
   subroutine sort_pp (subevt, en1, en2, en0)
     type(subevt_t), intent(inout) :: subevt
     type(eval_node_t), intent(in) :: en1, en2
     type(eval_node_t), intent(inout), optional :: en0
     integer, dimension(:), allocatable :: ival
     real(default), dimension(:), allocatable :: rval
     integer :: i, n1
     n1 = subevt_get_length (en1%pval)
     if (present (en0)) then
        select case (en0%result_type)
        case (V_INT);  allocate (ival (n1))
        case (V_REAL); allocate (rval (n1))
        end select
        do i = 1, n1
           en0%index = i
           en0%prt1 = subevt_get_prt (en1%pval, i)
           en0%prt2 = subevt_get_prt (en2%pval, 1)
           call eval_node_evaluate (en0)
           select case (en0%result_type)
           case (V_INT);  ival(i) = en0%ival
           case (V_REAL); rval(i) = en0%rval
           end select
        end do
        select case (en0%result_type)
        case (V_INT);  call subevt_sort (subevt, en1%pval, ival)
        case (V_REAL); call subevt_sort (subevt, en1%pval, rval)
        end select
```

```
      else
          call subevt_sort (subevt, en1%pval)
      end if
   end subroutine sort_pp
```

The following functions return a logical value. `all` evaluates to true if the condition `en0` is true for all valid element pairs of both subevents. Invalid pairs (with common `src` entry) are ignored.

   `any` and `no` are analogous.

⟨*Expressions: procedures*⟩+≡

```
   function all_pp (en1, en2, en0) result (lval)
     logical :: lval
     type(eval_node_t), intent(in) :: en1, en2
     type(eval_node_t), intent(inout) :: en0
     integer :: i, j, n1, n2
     n1 = subevt_get_length (en1%pval)
     n2 = subevt_get_length (en2%pval)
     lval = .true.
     LOOP1: do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = subevt_get_prt (en2%pval, j)
           if (are_disjoint (en0%prt1, en0%prt2)) then
              call eval_node_evaluate (en0)
              lval = en0%lval
              if (.not. lval)  exit LOOP1
           end if
        end do
     end do LOOP1
   end function all_pp

   function any_pp (en1, en2, en0) result (lval)
     logical :: lval
     type(eval_node_t), intent(in) :: en1, en2
     type(eval_node_t), intent(inout) :: en0
     integer :: i, j, n1, n2
     n1 = subevt_get_length (en1%pval)
     n2 = subevt_get_length (en2%pval)
     lval = .false.
     LOOP1: do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = subevt_get_prt (en2%pval, j)
           if (are_disjoint (en0%prt1, en0%prt2)) then
              call eval_node_evaluate (en0)
              lval = en0%lval
              if (lval)  exit LOOP1
           end if
        end do
     end do LOOP1
   end function any_pp
```

```
function no_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  lval = .true.
  LOOP1: do i = 1, n1
     en0%index = i
     en0%prt1 = subevt_get_prt (en1%pval, i)
     do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
           call eval_node_evaluate (en0)
           lval = .not. en0%lval
           if (lval)  exit LOOP1
        end if
     end do
  end do LOOP1
end function no_pp
```

This function evaluates an observable for a pair of particles. From the two
particle lists, we take the first pair without **src** overlap. If there is no valid
pair, we revert the status of the value to unknown.

⟨*Expressions: procedures*⟩+≡
```
subroutine eval_pp (en1, en2, en0, rval, is_known)
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  real(default), intent(out) :: rval
  logical, intent(out) :: is_known
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  rval = 0
  is_known = .false.
  LOOP1: do i = 1, n1
     en0%index = i
     en0%prt1 = subevt_get_prt (en1%pval, i)
     do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
           call eval_node_evaluate (en0)
           rval = en0%rval
           is_known = .true.
           exit LOOP1
        end if
     end do
  end do LOOP1
end subroutine eval_pp
```

This is the interface to user-supplied observables. The node **en0** evaluates to a

string that indicates the procedure name. We search for the procedure in the dynamic library and load it into the procedure pointer which is then called. `en1` is the subevent on which the external code operates. The external function returns a `c_int`, which we translate into a real value.

⟨*Expressions: procedures*⟩+≡

```
function user_obs_int_pp (en0, prt1, prt2) result (ival)
  integer :: ival
  type(eval_node_t), intent(inout) :: en0
  type(prt_t), intent(in) :: prt1, prt2
  type(string_t) :: name
  procedure(user_obs_int_binary), pointer :: user_obs
  call eval_node_evaluate (en0)
  if (en0%value_is_known) then
      select case (en0%result_type)
      case (V_STR);  name = en0%sval
      case default
         call msg_bug ("user_obs: procedure name must be a string")
         name = ""
      end select
      call c_f_procpointer (user_code_find_proc (name), user_obs)
      ival = user_obs (c_prt (prt1), c_prt (prt2))
  else
      call eval_node_write_rec (en0)
      call msg_fatal ("User observable name is undefined")
  end if
end function user_obs_int_pp

function user_obs_real_pp (en0, prt1, prt2) result (rval)
  real(default) :: rval
  type(eval_node_t), intent(inout) :: en0
  type(prt_t), intent(in) :: prt1, prt2
  type(string_t) :: name
  procedure(user_obs_real_binary), pointer :: user_obs
  call eval_node_evaluate (en0)
  if (en0%value_is_known) then
      select case (en0%result_type)
      case (V_STR);  name = en0%sval
      case default
         call msg_bug ("user_obs: procedure name must be a string")
         name = ""
      end select
      call c_f_procpointer (user_code_find_proc (name), user_obs)
      rval = user_obs (c_prt (prt1), c_prt (prt2))
  else
      call eval_node_write_rec (en0)
      call msg_fatal ("User observable name is undefined")
  end if
end function user_obs_real_pp
```

The following function returns an integer value, namely the number of valid particle-pairs from both lists for which the condition is true. Invalid pairs (with common `src` entry) are ignored. If there is no condition, it returns the number of valid particle pairs.

A function would be more natural. Making it a subroutine avoids another compiler bug (internal error in nagfor 5.2 (649)). (See the interface `binary_num`.)

⟨*Expressions: procedures*⟩+≡

```
subroutine count_pp (ival, en1, en2, en0)
  integer, intent(out) :: ival
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  integer :: i, j, n1, n2, count
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  if (present (en0)) then
     count = 0
     do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = subevt_get_prt (en2%pval, j)
           if (are_disjoint (en0%prt1, en0%prt2)) then
              call eval_node_evaluate (en0)
              if (en0%lval)  count = count + 1
           end if
        end do
     end do
  else
     count = 0
     do i = 1, n1
        do j = 1, n2
           if (are_disjoint (subevt_get_prt (en1%pval, i), &
                             subevt_get_prt (en2%pval, j))) then
              count = count + 1
           end if
        end do
     end do
  end if
  ival = count
end subroutine count_pp
```

This function makes up a subevent from the second argument which consists only of particles which match the PDG code array (first argument).

⟨*Expressions: procedures*⟩+≡

```
subroutine select_pdg_ca (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  if (present (en0)) then
     call subevt_select_pdg_code (subevt, en1%aval, en2%pval, en0%ival)
  else
     call subevt_select_pdg_code (subevt, en1%aval, en2%pval)
  end if
end subroutine select_pdg_ca
```

Binary string functions

Currently, the only string operation is concatenation.

⟨*Expressions: procedures*⟩+≡
```
    subroutine concat_ss (string, en1, en2)
      type(string_t), intent(out) :: string
      type(eval_node_t), intent(in) :: en1, en2
      string = en1%sval // en2%sval
    end subroutine concat_ss
```


5.7.4   Compiling the parse tree

The evaluation tree is built recursively by following a parse tree. Debug option:

⟨*Expressions: variables*⟩≡
```
    logical, parameter :: debug = .false.
```

Evaluate an expression. The requested type is given as an optional argument;
default is numeric (integer or real).

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_genexpr &
        (en, pn, var_list, result_type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in), optional :: result_type
      if (debug) then
         print *, "read genexpr";  call parse_node_write (pn)
      end if
      if (present (result_type)) then
         select case (result_type)
         case (V_INT, V_REAL, V_CMPLX)
            call eval_node_compile_expr  (en, pn, var_list)
         case (V_LOG)
            call eval_node_compile_lexpr (en, pn, var_list)
         case (V_SEV)
            call eval_node_compile_pexpr (en, pn, var_list)
         case (V_PDG)
            call eval_node_compile_cexpr (en, pn, var_list)
         case (V_STR)
            call eval_node_compile_sexpr (en, pn, var_list)
         end select
      else
         call eval_node_compile_expr  (en, pn, var_list)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done genexpr"
      end if
    end subroutine eval_node_compile_genexpr
```

Numeric expressions

This procedure compiles a numerical expression. This is a single term or a sum
or difference of terms. We have to account for all combinations of integer and
real arguments. If both are constant, we immediately do the calculation and
allocate a constant node.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_expr (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_term, pn_addition, pn_op, pn_arg
      type(eval_node_t), pointer :: en1, en2
      type(string_t) :: key
      integer :: t1, t2, t
      if (debug) then
         print *, "read expr";  call parse_node_write (pn)
      end if
      pn_term => parse_node_get_sub_ptr (pn)
      select case (char (parse_node_get_rule_key (pn_term)))
      case ("term")
         call eval_node_compile_term (en, pn_term, var_list)
         pn_addition => parse_node_get_next_ptr (pn_term, tag="addition")
      case ("addition")
         en => null ()
         pn_addition => pn_term
      case default
         call parse_node_mismatch ("term|addition", pn)
      end select
      do while (associated (pn_addition))
         pn_op => parse_node_get_sub_ptr (pn_addition)
         pn_arg => parse_node_get_next_ptr (pn_op, tag="term")
         call eval_node_compile_term (en2, pn_arg, var_list)
         t2 = en2%result_type
         if (associated (en)) then
            en1 => en
            t1 = en1%result_type
         else
            allocate (en1)
            select case (t2)
            case (V_INT);  call eval_node_init_int  (en1, 0)
            case (V_REAL); call eval_node_init_real (en1, 0._default)
            case (V_CMPLX); call eval_node_init_cmplx (en1, cmplx &
                           (0._default, 0._default, kind=default))
            end select
            t1 = t2
         end if
         t = numeric_result_type (t1, t2)
         allocate (en)
         key = parse_node_get_key (pn_op)
         if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            select case (char (key))
            case ("+")
               select case (t1)
```

```
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int   (en, add_ii (en1, en2))
         case (V_REAL); call eval_node_init_real  (en, add_ir (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, add_ic (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real  (en, add_ri (en1, en2))
         case (V_REAL); call eval_node_init_real  (en, add_rr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, add_rc (en1, en2))
         end select
       case (V_CMPLX)
         select case (t2)
         case (V_INT);  call eval_node_init_cmplx (en, add_ci (en1, en2))
         case (V_REAL); call eval_node_init_cmplx (en, add_cr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, add_cc (en1, en2))
         end select
      end select
   case ("-")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int   (en, sub_ii (en1, en2))
         case (V_REAL); call eval_node_init_real  (en, sub_ir (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, sub_ic (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real  (en, sub_ri (en1, en2))
         case (V_REAL); call eval_node_init_real  (en, sub_rr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, sub_rc (en1, en2))
         end select
      case (V_CMPLX)
         select case (t2)
         case (V_INT);  call eval_node_init_cmplx (en, sub_ci (en1, en2))
         case (V_REAL); call eval_node_init_cmplx (en, sub_cr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, sub_cc (en1, en2))
         end select
      end select
   end select
   call eval_node_final_rec (en1)
   call eval_node_final_rec (en2)
   deallocate (en1, en2)
else
   call eval_node_init_branch (en, key, t, en1, en2)
   select case (char (key))
   case ("+")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_int   (en, add_ii)
         case (V_REAL); call eval_node_set_op2_real  (en, add_ir)
         case (V_CMPLX); call eval_node_set_op2_cmplx (en, add_ic)
```

```
                        end select
                case (V_REAL)
                   select case (t2)
                   case (V_INT);    call eval_node_set_op2_real (en, add_ri)
                   case (V_REAL);   call eval_node_set_op2_real (en, add_rr)
                   case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_rc)
                   end select
                case (V_CMPLX)
                   select case (t2)
                   case (V_INT);    call eval_node_set_op2_cmplx (en, add_ci)
                   case (V_REAL);   call eval_node_set_op2_cmplx (en, add_cr)
                   case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_cc)
                   end select
                end select
             case ("-")
                select case (t1)
                case (V_INT)
                   select case (t2)
                   case (V_INT);    call eval_node_set_op2_int  (en, sub_ii)
                   case (V_REAL);   call eval_node_set_op2_real (en, sub_ir)
                   case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_ic)
                   end select
                case (V_REAL)
                   select case (t2)
                   case (V_INT);    call eval_node_set_op2_real (en, sub_ri)
                   case (V_REAL);   call eval_node_set_op2_real (en, sub_rr)
                   case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_rc)
                   end select
                case (V_CMPLX)
                   select case (t2)
                   case (V_INT);    call eval_node_set_op2_cmplx (en, sub_ci)
                   case (V_REAL);   call eval_node_set_op2_cmplx (en, sub_cr)
                   case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_cc)
                   end select
                end select
             end select
          end if
          pn_addition => parse_node_get_next_ptr (pn_addition)
       end do
       if (debug) then
          call eval_node_write (en)
          print *, "done expr"
       end if
     end subroutine eval_node_compile_expr
```

⟨*Expressions: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_term (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_factor, pn_multiplication, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: key
    integer :: t1, t2, t
```

```
if (debug) then
   print *, "read term";  call parse_node_write (pn)
end if
pn_factor => parse_node_get_sub_ptr (pn, tag="factor")
call eval_node_compile_factor (en, pn_factor, var_list)
pn_multiplication => &
     parse_node_get_next_ptr (pn_factor, tag="multiplication")
do while (associated (pn_multiplication))
   pn_op => parse_node_get_sub_ptr (pn_multiplication)
   pn_arg => parse_node_get_next_ptr (pn_op, tag="factor")
   en1 => en
   call eval_node_compile_factor (en2, pn_arg, var_list)
   t1 = en1%result_type
   t2 = en2%result_type
   t = numeric_result_type (t1, t2)
   allocate (en)
   key = parse_node_get_key (pn_op)
   if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
      select case (char (key))
      case ("*")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);   call eval_node_init_int   (en, mul_ii (en1, en2))
            case (V_REAL);  call eval_node_init_real  (en, mul_ir (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, mul_ic (en1, en2))
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);   call eval_node_init_real  (en, mul_ri (en1, en2))
            case (V_REAL);  call eval_node_init_real  (en, mul_rr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, mul_rc (en1, en2))
            end select
          case (V_CMPLX)
            select case (t2)
            case (V_INT);   call eval_node_init_cmplx (en, mul_ci (en1, en2))
            case (V_REAL);  call eval_node_init_cmplx (en, mul_cr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, mul_cc (en1, en2))
            end select
         end select
      case ("/")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);   call eval_node_init_int   (en, div_ii (en1, en2))
            case (V_REAL);  call eval_node_init_real  (en, div_ir (en1, en2))
            case (V_CMPLX); call eval_node_init_real  (en, div_ir (en1, en2))
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);   call eval_node_init_real  (en, div_ri (en1, en2))
            case (V_REAL);  call eval_node_init_real  (en, div_rr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, div_rc (en1, en2))
            end select
```

```
               case (V_CMPLX)
                  select case (t2)
                  case (V_INT);   call eval_node_init_cmplx (en, div_ci (en1, en2))
                  case (V_REAL);  call eval_node_init_cmplx (en, div_cr (en1, en2))
                  case (V_CMPLX); call eval_node_init_cmplx (en, div_cc (en1, en2))
                  end select
               end select
            end select
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
         else
            call eval_node_init_branch (en, key, t, en1, en2)
            select case (char (key))
            case ("*")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);   call eval_node_set_op2_int   (en, mul_ii)
                  case (V_REAL);  call eval_node_set_op2_real  (en, mul_ir)
                  case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_ic)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);   call eval_node_set_op2_real  (en, mul_ri)
                  case (V_REAL);  call eval_node_set_op2_real  (en, mul_rr)
                  case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_rc)
                  end select
               case (V_CMPLX)
                  select case (t2)
                  case (V_INT);   call eval_node_set_op2_cmplx (en, mul_ci)
                  case (V_REAL);  call eval_node_set_op2_cmplx (en, mul_cr)
                  case (V_CMPLX); call eval_node_set_op2_cmplx (en, mul_cc)
                  end select
               end select
            case ("/")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);   call eval_node_set_op2_int   (en, div_ii)
                  case (V_REAL);  call eval_node_set_op2_real  (en, div_ir)
                  case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_ic)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);   call eval_node_set_op2_real  (en, div_ri)
                  case (V_REAL);  call eval_node_set_op2_real  (en, div_rr)
                  case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_rc)
                  end select
               case (V_CMPLX)
                  select case (t2)
                  case (V_INT);   call eval_node_set_op2_cmplx (en, div_ci)
                  case (V_REAL);  call eval_node_set_op2_cmplx (en, div_cr)
                  case (V_CMPLX); call eval_node_set_op2_cmplx (en, div_cc)
```

```
                 end select
              end select
           end select
        end if
        pn_multiplication => parse_node_get_next_ptr (pn_multiplication)
      end do
      if (debug) then
         call eval_node_write (en)
         print *, "done term"
      end if
    end subroutine eval_node_compile_term
```

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_factor (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_value, pn_exponentiation, pn_op, pn_arg
      type(eval_node_t), pointer :: en1, en2
      type(string_t) :: key
      integer :: t1, t2, t
      if (debug) then
         print *, "read factor";  call parse_node_write (pn)
      end if
      pn_value => parse_node_get_sub_ptr (pn)
      call eval_node_compile_signed_value (en, pn_value, var_list)
      pn_exponentiation => &
           parse_node_get_next_ptr (pn_value, tag="exponentiation")
      if (associated (pn_exponentiation)) then
         pn_op => parse_node_get_sub_ptr (pn_exponentiation)
         pn_arg => parse_node_get_next_ptr (pn_op)
         en1 => en
         call eval_node_compile_signed_value (en2, pn_arg, var_list)
         t1 = en1%result_type
         t2 = en2%result_type
         t = numeric_result_type (t1, t2)
         allocate (en)
         key = parse_node_get_key (pn_op)
         if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            select case (t1)
            case (V_INT)
               select case (t2)
               case (V_INT);   call eval_node_init_int   (en, pow_ii (en1, en2))
               case (V_REAL);  call eval_node_init_real  (en, pow_ir (en1, en2))
               case (V_CMPLX); call eval_node_init_cmplx (en, pow_ic (en1, en2))
               end select
            case (V_REAL)
               select case (t2)
               case (V_INT);   call eval_node_init_real  (en, pow_ri (en1, en2))
               case (V_REAL);  call eval_node_init_real  (en, pow_rr (en1, en2))
               case (V_CMPLX); call eval_node_init_cmplx (en, pow_rc (en1, en2))
               end select
            case (V_CMPLX)
               select case (t2)
```

```
                          case (V_INT);   call eval_node_init_cmplx (en, pow_ci (en1, en2))
                          case (V_REAL);  call eval_node_init_cmplx (en, pow_cr (en1, en2))
                          case (V_CMPLX); call eval_node_init_cmplx (en, pow_cc (en1, en2))
                          end select
                       end select
                       call eval_node_final_rec (en1)
                       call eval_node_final_rec (en2)
                       deallocate (en1, en2)
                    else
                       call eval_node_init_branch (en, key, t, en1, en2)
                       select case (t1)
                       case (V_INT)
                          select case (t2)
                          case (V_INT);   call eval_node_set_op2_int  (en, pow_ii)
                          case (V_REAL,V_CMPLX);  call eval_type_error (pn, "exponentiation", t1)
                          end select
                       case (V_REAL)
                          select case (t2)
                          case (V_INT);   call eval_node_set_op2_real (en, pow_ri)
                          case (V_REAL);  call eval_node_set_op2_real (en, pow_rr)
                          case (V_CMPLX);  call eval_type_error (pn, "exponentiation", t1)
                          end select
                       case (V_CMPLX)
                          select case (t2)
                          case (V_INT);   call eval_node_set_op2_cmplx (en, pow_ci)
                          case (V_REAL);  call eval_node_set_op2_cmplx (en, pow_cr)
                          case (V_CMPLX);  call eval_node_set_op2_cmplx (en, pow_cc)
                          end select
                       end select
                    end if
                 end if
                 if (debug) then
                    call eval_node_write (en)
                    print *, "done factor"
                 end if
              end subroutine eval_node_compile_factor


⟨Expressions: procedures⟩+≡
          recursive subroutine eval_node_compile_signed_value (en, pn, var_list)
            type(eval_node_t), pointer :: en
            type(parse_node_t), intent(in) :: pn
            type(var_list_t), intent(in), target :: var_list
            type(parse_node_t), pointer :: pn_arg
            type(eval_node_t), pointer :: en1
            integer :: t
            if (debug) then
               print *, "read signed value";  call parse_node_write (pn)
            end if
            select case (char (parse_node_get_rule_key (pn)))
            case ("signed_value")
               pn_arg => parse_node_get_sub_ptr (pn, 2)
               call eval_node_compile_value (en1, pn_arg, var_list)
               t = en1%result_type
               allocate (en)
```

```
      if (en1%type == EN_CONSTANT) then
         select case (t)
         case (V_INT);   call eval_node_init_int   (en, neg_i (en1))
         case (V_REAL);  call eval_node_init_real  (en, neg_r (en1))
         case (V_CMPLX); call eval_node_init_cmplx (en, neg_c (en1))
         end select
         call eval_node_final_rec (en1)
         deallocate (en1)
      else
         call eval_node_init_branch (en, var_str ("-"), t, en1)
         select case (t)
         case (V_INT);   call eval_node_set_op1_int   (en, neg_i)
         case (V_REAL);  call eval_node_set_op1_real  (en, neg_r)
         case (V_CMPLX); call eval_node_set_op1_cmplx (en, neg_c)
         end select
      end if
   case default
      call eval_node_compile_value (en, pn, var_list)
   end select
   if (debug) then
      call eval_node_write (en)
      print *, "done signed value"
   end if
 end subroutine eval_node_compile_signed_value
```

Integer, real and complex values have an optional unit. The unit is extracted
and applied immediately. An integer with unit evaluates to a real constant.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_value (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     if (debug) then
        print *, "read value";  call parse_node_write (pn)
     end if
     select case (char (parse_node_get_rule_key (pn)))
     case ("integer_value", "real_value", "complex_value")
        call eval_node_compile_numeric_value (en, pn)
     case ("pi")
        call eval_node_compile_constant (en, pn)
     case ("I")
        call eval_node_compile_constant (en, pn)
     case ("variable")
        call eval_node_compile_variable (en, pn, var_list)
     case ("result")
        call eval_node_compile_result (en, pn, var_list)
     case ("user_observable")
        call eval_node_compile_user_observable (en, pn, var_list)
     case ("expr")
        call eval_node_compile_expr (en, pn, var_list)
     case ("block_expr")
        call eval_node_compile_block_expr (en, pn, var_list)
     case ("conditional_expr")
```

```
      call eval_node_compile_conditional (en, pn, var_list)
   case ("unary_function")
      call eval_node_compile_unary_function (en, pn, var_list)
   case ("binary_function")
      call eval_node_compile_binary_function (en, pn, var_list)
   case ("eval_fun")
      call eval_node_compile_eval_function (en, pn, var_list)
   case ("count_fun", "user_event_fun")
      call eval_node_compile_numeric_function (en, pn, var_list)
   case default
      call parse_node_mismatch &
            ("integer|real|complex|constant|variable|" // &
             "expr|block_expr|conditional_expr|" // &
             "unary_function|binary_function|numeric_pexpr", pn)
   end select
   if (debug) then
      call eval_node_write (en)
      print *, "done value"
   end if
end subroutine eval_node_compile_value
```

Real, complex and integer values are numeric literals with an optional unit
attached. In case of an integer, the unit actually makes it a real value in
disguise. The signed version of real values is not possible in generic expressions;
it is a special case for numeric constants in model files (see below). We do not
introduce signed versions of complex values.

⟨*Expressions: procedures*⟩+≡
```
   subroutine eval_node_compile_numeric_value (en, pn)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in), target :: pn
     type(parse_node_t), pointer :: pn_val, pn_unit
     allocate (en)
     pn_val => parse_node_get_sub_ptr (pn)
     pn_unit => parse_node_get_next_ptr (pn_val)
     select case (char (parse_node_get_rule_key (pn)))
     case ("integer_value")
        if (associated (pn_unit)) then
          call eval_node_init_real (en, &
               parse_node_get_integer (pn_val) * parse_node_get_unit (pn_unit))
        else
           call eval_node_init_int (en, parse_node_get_integer (pn_val))
        end if
     case ("real_value")
        if (associated (pn_unit)) then
          call eval_node_init_real (en, &
               parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
        else
           call eval_node_init_real (en, parse_node_get_real (pn_val))
        end if
     case ("complex_value")
        if (associated (pn_unit)) then
          call eval_node_init_cmplx (en, &
               parse_node_get_cmplx (pn_val) * parse_node_get_unit (pn_unit))
```

```
      else
         call eval_node_init_cmplx (en, parse_node_get_cmplx (pn_val))
      end if
   case ("neg_real_value")
      pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
      pn_unit => parse_node_get_next_ptr (pn_val)
      if (associated (pn_unit)) then
         call eval_node_init_real (en, &
              - parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
      else
         call eval_node_init_real (en, - parse_node_get_real (pn_val))
      end if
   case ("pos_real_value")
      pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
      pn_unit => parse_node_get_next_ptr (pn_val)
      if (associated (pn_unit)) then
         call eval_node_init_real (en, &
              parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
      else
         call eval_node_init_real (en, parse_node_get_real (pn_val))
      end if
   case default
      call parse_node_mismatch &
      ("integer_value|real_value|complex_value|neg_real_value|pos_real_value", pn)
   end select
 end subroutine eval_node_compile_numeric_value
```

These are the units, predefined and hardcoded. The default energy unit is
GeV, the default angular unit is radians. We include units for observables of
dimension energy squared. Luminosities are normalized in inverse femtobarns.

⟨*Expressions: procedures*⟩+≡

```
   function parse_node_get_unit (pn) result (factor)
     real(default) :: factor
     real(default) :: unit
     type(parse_node_t), intent(in) :: pn
     type(parse_node_t), pointer :: pn_unit, pn_unit_power
     type(parse_node_t), pointer :: pn_frac, pn_num, pn_int, pn_div, pn_den
     integer :: num, den
     pn_unit => parse_node_get_sub_ptr (pn)
     select case (char (parse_node_get_key (pn_unit)))
     case ("TeV");   unit = 1.e3_default
     case ("GeV");   unit = 1
     case ("MeV");   unit = 1.e-3_default
     case ("keV");   unit = 1.e-6_default
     case ("eV");    unit = 1.e-9_default
     case ("meV");    unit = 1.e-12_default
     case ("nbarn");  unit = 1.e6_default
     case ("pbarn");  unit = 1.e3_default
     case ("fbarn");  unit = 1
     case ("abarn");  unit = 1.e-3_default
     case ("rad");     unit = 1
     case ("mrad");     unit = 1.e-3_default
     case ("degree");  unit = degree
```

```
            case ("%");   unit = 1.e-2_default
            case default
               call msg_bug (" Unit '" // &
                     char (parse_node_get_key (pn)) // "' is undefined.")
            end select
            pn_unit_power => parse_node_get_next_ptr (pn_unit)
            if (associated (pn_unit_power)) then
               pn_frac => parse_node_get_sub_ptr (pn_unit_power, 2)
               pn_num => parse_node_get_sub_ptr (pn_frac)
               select case (char (parse_node_get_rule_key (pn_num)))
               case ("neg_int")
                  pn_int => parse_node_get_sub_ptr (pn_num, 2)
                  num = - parse_node_get_integer (pn_int)
               case ("pos_int")
                  pn_int => parse_node_get_sub_ptr (pn_num, 2)
                  num = parse_node_get_integer (pn_int)
               case ("integer_literal")
                  num = parse_node_get_integer (pn_num)
               case default
                  call parse_node_mismatch ("neg_int|pos_int|integer_literal", pn_num)
               end select
               pn_div => parse_node_get_next_ptr (pn_num)
               if (associated (pn_div)) then
                  pn_den => parse_node_get_sub_ptr (pn_div, 2)
                  den = parse_node_get_integer (pn_den)
               else
                  den = 1
               end if
            else
               num = 1
               den = 1
            end if
            factor = unit ** (real (num, default) / den)
         end function parse_node_get_unit
```

There are only two predefined constants, but more can be added easily.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_compile_constant (en, pn)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      if (debug) then
         print *, "read constant";  call parse_node_write (pn)
      end if
      allocate (en)
      select case (char (parse_node_get_key (pn)))
      case ("pi");     call eval_node_init_real (en, pi)
      case ("I");      call eval_node_init_cmplx (en, imago)
      case default
         call parse_node_mismatch ("pi or I", pn)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done constant"
      end if
```

```
      end subroutine eval_node_compile_constant
```

Compile a variable, with or without a specified type. Take the list of variables, look for the name and make a node with a pointer to the value. If no type is provided, the variable is numeric, and the stored value determines whether it is real or integer.

We explicitly demand that the variable is defined, so we do not accidentally point to variables that are declared only later in the script but have come into existence in a previous compilation pass.

Variables may actually be anonymous, these are expressions in disguise. In that case, the expression replaces the variable name in the parse tree, and we allocate an ordinary expression node in the eval tree.

Variables of type `V_PDG` (pdg-code array) are not treated here. They are handled by `eval_node_compile_cvariable`.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_variable (en, pn, var_list, var_type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in), optional :: var_type
      type(parse_node_t), pointer :: pn_name
      type(string_t) :: var_name
      type(var_entry_t), pointer :: var
      logical, target, save :: no_lval
      real(default), target, save :: no_rval
      type(subevt_t), target, save :: no_pval
      type(string_t), target, save :: no_sval
      logical, target, save :: unknown = .false.
      if (debug) then
         print *, "read variable";  call parse_node_write (pn)
      end if
      if (present (var_type)) then
         select case (var_type)
         case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
                  V_OBS2_INT, V_CMPLX)
           pn_name => pn
         case default
           pn_name => parse_node_get_sub_ptr (pn, 2)
         end select
      else
         pn_name => pn
      end if
      select case (char (parse_node_get_rule_key (pn_name)))
      case ("expr")
         call eval_node_compile_expr (en, pn_name, var_list)
      case ("lexpr")
         call eval_node_compile_lexpr (en, pn_name, var_list)
      case ("sexpr")
         call eval_node_compile_sexpr (en, pn_name, var_list)
      case ("pexpr")
         call eval_node_compile_pexpr (en, pn_name, var_list)
      case ("variable")
         var_name = parse_node_get_string (pn_name)
```

```
if (present (var_type)) then
   select case (var_type)
   case (V_LOG);  var_name = "?" // var_name
   case (V_SEV);  var_name = "@" // var_name
   case (V_STR);  var_name = "$" // var_name   ! $ sign
   end select
end if
var => var_list_get_var_ptr &
     (var_list, var_name, var_type, defined=.true.)
allocate (en)
if (associated (var)) then
   select case (var_entry_get_type (var))
   case (V_LOG)
      call eval_node_init_log_ptr &
           (en, var_entry_get_name (var), var_entry_get_lval_ptr (var), &
            var_entry_get_known_ptr (var))
   case (V_INT)
      call eval_node_init_int_ptr &
           (en, var_entry_get_name (var), var_entry_get_ival_ptr (var), &
            var_entry_get_known_ptr (var))
   case (V_REAL)
      call eval_node_init_real_ptr &
           (en, var_entry_get_name (var), var_entry_get_rval_ptr (var), &
            var_entry_get_known_ptr (var))
   case (V_CMPLX)
      call eval_node_init_cmplx_ptr &
           (en, var_entry_get_name (var), var_entry_get_cval_ptr (var), &
            var_entry_get_known_ptr (var))
   case (V_SEV)
      call eval_node_init_subevt_ptr &
           (en, var_entry_get_name (var), var_entry_get_pval_ptr (var), &
            var_entry_get_known_ptr (var))
   case (V_STR)
      call eval_node_init_string_ptr &
           (en, var_entry_get_name (var), var_entry_get_sval_ptr (var), &
            var_entry_get_known_ptr (var))
   case (V_OBS1_INT, V_OBS2_INT, V_OBS1_REAL, V_OBS2_REAL)
      call eval_node_init_obs  (en, var)
   case default
      call parse_node_write (pn)
      call msg_fatal ("Variable of this type " // &
           "is not allowed in the present context")
      if (present (var_type)) then
         select case (var_type)
         case (V_LOG)
            call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
         case (V_SEV)
            call eval_node_init_subevt_ptr &
                 (en, var_name, no_pval, unknown)
         case (V_STR)
            call eval_node_init_string_ptr &
                 (en, var_name, no_sval, unknown)
         end select
      else
```

```
                    call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
                end if
            end select
        else
            call parse_node_write (pn)
            call msg_error ("This variable is undefined at this point")
            if (present (var_type)) then
                select case (var_type)
                case (V_LOG)
                    call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
                case (V_SEV)
                    call eval_node_init_subevt_ptr &
                        (en, var_name, no_pval, unknown)
                case (V_STR)
                    call eval_node_init_string_ptr (en, var_name, no_sval, unknown)
                end select
            else
                call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
            end if
        end if
    end select
    if (debug) then
        call eval_node_write (en)
        print *, "done variable"
    end if
end subroutine eval_node_compile_variable
```

In a given context, a variable has to have a certain type.
⟨*Expressions: procedures*⟩+≡
```
    subroutine check_var_type (pn, ok, type_actual, type_requested)
      type(parse_node_t), intent(in) :: pn
      logical, intent(out) :: ok
      integer, intent(in) :: type_actual
      integer, intent(in), optional :: type_requested
      if (present (type_requested)) then
          select case (type_requested)
          case (V_LOG)
              select case (type_actual)
              case (V_LOG)
              case default
                 call parse_node_write (pn)
                 call msg_fatal ("Variable type is invalid (should be logical)")
                 ok = .false.
              end select
          case (V_SEV)
              select case (type_actual)
              case (V_SEV)
              case default
                 call parse_node_write (pn)
                 call msg_fatal &
                      ("Variable type is invalid (should be particle set)")
                 ok = .false.
              end select
          case (V_PDG)
```

```
      select case (type_actual)
      case (V_PDG)
      case default
         call parse_node_write (pn)
         call msg_fatal &
              ("Variable type is invalid (should be PDG array)")
         ok = .false.
      end select
   case (V_STR)
      select case (type_actual)
      case (V_STR)
      case default
         call parse_node_write (pn)
         call msg_fatal &
              ("Variable type is invalid (should be string)")
         ok = .false.
      end select
   case default
      call parse_node_write (pn)
      call msg_bug ("Variable type is unknown")
   end select
 else
   select case (type_actual)
   case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
           V_OBS2_INT, V_CMPLX)
   case default
      call parse_node_write (pn)
      call msg_fatal ("Variable type is invalid (should be numeric)")
      ok = .false.
   end select
 end if
 ok = .true.
end subroutine check_var_type
```

Retrieve the result of an integration. If the requested process has been integrated, the results are available as special variables. (The variables cannot be accessed in the usual way since they contain brackets in their names.)

Since this compilation step may occur before the processes have been loaded, we have to initialize the required variables before they are used.

⟨*Expressions: procedures*⟩+≡
```
  subroutine eval_node_compile_result (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in), target :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_key, pn_prc_id
    type(string_t) :: key, prc_id, var_name
    type(var_entry_t), pointer :: var
    if (debug) then
       print *, "read result";  call parse_node_write (pn)
    end if
    pn_key => parse_node_get_sub_ptr (pn)
    pn_prc_id => parse_node_get_next_ptr (pn_key)
    key = parse_node_get_key (pn_key)
```

```
        prc_id = parse_node_get_string (pn_prc_id)
        var_name = key // "(" // prc_id // ")"
        var => var_list_get_var_ptr (var_list, var_name)
        if (associated (var)) then
           allocate (en)
           select case (char(key))
           case ("num_id", "n_calls")
              call eval_node_init_int_ptr &
                   (en, var_name, var_entry_get_ival_ptr (var), &
                   var_entry_get_known_ptr (var))
           case ("integral", "error")
              call eval_node_init_real_ptr &
                   (en, var_name, var_entry_get_rval_ptr (var), &
                 var_entry_get_known_ptr (var))
           end select
        else
           call msg_fatal ("Result variable '" // char (var_name) &
                // "' is undefined (call 'integrate' before use)")
        end if
        if (debug) then
           call eval_node_write (en)
           print *, "done result"
        end if
     end subroutine eval_node_compile_result
```

This user observable behaves like a variable. We link the node to the generic user-observable entry in the variable list. The syntax element has an argument which provides the name of the user variable, this is stored as an eval-node alongside with the variable. When the variable value is used, the user-supplied external function is called and provides the (real) result value.

⟨*Expressions: procedures*⟩+≡
```
     subroutine eval_node_compile_user_observable (en, pn, var_list)
       type(eval_node_t), pointer :: en
       type(parse_node_t), intent(in), target :: pn
       type(var_list_t), intent(in), target :: var_list
       type(parse_node_t), pointer :: pn_key, pn_arg, pn_obs
       type(eval_node_t), pointer :: en0
       integer :: res_type
       type(string_t) :: var_name
       type(var_entry_t), pointer :: var
       if (debug) then
          print *, "read user observable";  call parse_node_write (pn)
       end if
       pn_key => parse_node_get_sub_ptr (pn)
       select case (char (parse_node_get_key (pn_key)))
       case ("user_obs")
          res_type = V_REAL
       case default
          call parse_node_write (pn_key)
          call msg_bug ("user_observable: wrong keyword")
       end select
       pn_arg => parse_node_get_next_ptr (pn_key)
       pn_obs => parse_node_get_sub_ptr (pn_arg)
```

```
      call eval_node_compile_sexpr (en0, pn_obs, var_list)
      select case (res_type)
      case (V_INT);  var_name = "_User_obs_int"
      case (V_REAL); var_name = "_User_obs_real"
      end select
      var => var_list_get_var_ptr (var_list, var_name, defined=.true.)
      allocate (en)
      if (associated (var)) then
         call eval_node_init_obs  (en, var, en0)
      else
         call parse_node_write (pn)
         call msg_error ("This variable is undefined at this point")
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done user observable"
      end if
   end subroutine eval_node_compile_user_observable
```

Functions with a single argument. For non-constant arguments, watch for functions which convert their argument to a different type.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_unary_function (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_fname, pn_arg
     type(eval_node_t), pointer :: en1
     type(string_t) :: key
     integer :: t
     if (debug) then
        print *, "read unary function";  call parse_node_write (pn)
     end if
     pn_fname => parse_node_get_sub_ptr (pn)
     pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg1")
     call eval_node_compile_expr &
          (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
     t = en1%result_type
     allocate (en)
     key = parse_node_get_key (pn_fname)
     if (en1%type == EN_CONSTANT) then
        select case (char (key))
        case ("complex")
           select case (t)
           case (V_INT);  call eval_node_init_cmplx (en, cmplx_i (en1))
           case (V_REAL); call eval_node_init_cmplx (en, cmplx_r (en1))
           case (V_CMPLX); deallocate (en);  en => en1;  en1 => null ()
           case default;  call eval_type_error (pn, char (key), t)
           end select
        case ("real")
           select case (t)
           case (V_INT);  call eval_node_init_real (en, real_i (en1))
           case (V_REAL); deallocate (en);  en => en1;  en1 => null ()
```

```
         case (V_CMPLX); call eval_node_init_real (en, real_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("int")
         select case (t)
         case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
         case (V_REAL); call eval_node_init_int  (en, int_r (en1))
         case (V_CMPLX); call eval_node_init_int  (en, int_c (en1))
         end select
      case ("nint")
         select case (t)
         case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
         case (V_REAL); call eval_node_init_int  (en, nint_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("floor")
         select case (t)
         case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
         case (V_REAL); call eval_node_init_int  (en, floor_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("ceiling")
         select case (t)
         case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
         case (V_REAL); call eval_node_init_int  (en, ceiling_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("abs")
         select case (t)
         case (V_INT);  call eval_node_init_int  (en, abs_i (en1))
         case (V_REAL); call eval_node_init_real (en, abs_r (en1))
         case (V_CMPLX); call eval_node_init_real (en, abs_c (en1))
         end select
      case ("sgn")
         select case (t)
         case (V_INT);  call eval_node_init_int  (en, sgn_i (en1))
         case (V_REAL); call eval_node_init_real (en, sgn_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("sqrt")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, sqrt_r (en1))
         case (V_CMPLX); call eval_node_init_cmplx (en, sqrt_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("exp")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, exp_r (en1))
         case (V_CMPLX); call eval_node_init_cmplx (en, exp_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("log")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, log_r (en1))
```

```
         case (V_CMPLX); call eval_node_init_cmplx (en, log_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("log10")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, log10_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("sin")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, sin_r (en1))
         case (V_CMPLX); call eval_node_init_cmplx (en, sin_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("cos")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, cos_r (en1))
         case (V_CMPLX); call eval_node_init_cmplx (en, cos_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("tan")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, tan_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("asin")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, asin_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("acos")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, acos_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("atan")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, atan_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("sinh")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, sinh_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("cosh")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, cosh_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("tanh")
         select case (t)
         case (V_REAL); call eval_node_init_real (en, tanh_r (en1))
         case default;  call eval_type_error (pn, char (key), t)
```

```
              end select
         case default
              call parse_node_mismatch ("function name", pn_fname)
         end select
         if (associated (en1)) then
              call eval_node_final_rec (en1)
              deallocate (en1)
         end if
    else
         select case (char (key))
         case ("complex")
              call eval_node_init_branch (en, key, V_CMPLX, en1)
         case ("real")
              call eval_node_init_branch (en, key, V_REAL, en1)
         case ("int", "nint", "floor", "ceiling")
              call eval_node_init_branch (en, key, V_INT, en1)
         case default
              call eval_node_init_branch (en, key, t, en1)
         end select
         select case (char (key))
         case ("complex")
              select case (t)
              case (V_INT);   call eval_node_set_op1_cmplx (en, cmplx_i)
              case (V_REAL); call eval_node_set_op1_cmplx (en, cmplx_r)
              case (V_CMPLX); deallocate (en);  en => en1
              case default; call eval_type_error (pn, char (key), t)
              end select
         case ("real")
              select case (t)
              case (V_INT);   call eval_node_set_op1_real (en, real_i)
              case (V_REAL); deallocate (en); en => en1
              case (V_CMPLX); call eval_node_set_op1_real (en, real_c)
              case default; call eval_type_error (pn, char (key), t)
              end select
         case ("int")
              select case (t)
              case (V_INT);   deallocate (en); en => en1
              case (V_REAL); call eval_node_set_op1_int (en, int_r)
              case (V_CMPLX); call eval_node_set_op1_int (en, int_c)
              end select
         case ("nint")
              select case (t)
              case (V_INT);   deallocate (en); en => en1
              case (V_REAL); call eval_node_set_op1_int (en, nint_r)
              case default; call eval_type_error (pn, char (key), t)
              end select
         case ("floor")
              select case (t)
              case (V_INT);   deallocate (en); en => en1
              case (V_REAL); call eval_node_set_op1_int (en, floor_r)
              case default; call eval_type_error (pn, char (key), t)
              end select
         case ("ceiling")
              select case (t)
```

```
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_set_op1_int (en, ceiling_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("abs")
      select case (t)
      case (V_INT);  call eval_node_set_op1_int  (en, abs_i)
      case (V_REAL); call eval_node_set_op1_real (en, abs_r)
      case (V_CMPLX);
         call eval_node_init_branch (en, key, V_REAL, en1)
         call eval_node_set_op1_real (en, abs_c)
      end select
   case ("sgn")
      select case (t)
      case (V_INT);  call eval_node_set_op1_int  (en, sgn_i)
      case (V_REAL); call eval_node_set_op1_real (en, sgn_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("sqrt")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, sqrt_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, sqrt_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("exp")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, exp_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, exp_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("log")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, log_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, log_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("log10")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, log10_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("sin")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, sin_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, sin_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("cos")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, cos_r)
      case (V_CMPLX); call eval_node_set_op1_cmplx (en, cos_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("tan")
```

```
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, tan_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("asin")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, asin_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("acos")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, acos_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("atan")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, atan_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("sinh")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, sinh_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("cosh")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, cosh_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("tanh")
      select case (t)
      case (V_REAL); call eval_node_set_op1_real (en, tanh_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case default
      call parse_node_mismatch ("function name", pn_fname)
   end select
 end if
 if (debug) then
    call eval_node_write (en)
    print *, "done function"
 end if
end subroutine eval_node_compile_unary_function
```

Functions with two arguments.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_binary_function (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_fname, pn_arg, pn_arg1, pn_arg2
  type(eval_node_t), pointer :: en1, en2
  type(string_t) :: key
  integer :: t1, t2
```

```
if (debug) then
   print *, "read binary function";  call parse_node_write (pn)
end if
pn_fname => parse_node_get_sub_ptr (pn)
pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg2")
pn_arg1 => parse_node_get_sub_ptr (pn_arg, tag="expr")
pn_arg2 => parse_node_get_next_ptr (pn_arg1, tag="expr")
call eval_node_compile_expr (en1, pn_arg1, var_list)
call eval_node_compile_expr (en2, pn_arg2, var_list)
t1 = en1%result_type
t2 = en2%result_type
allocate (en)
key = parse_node_get_key (pn_fname)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (char (key))
   case ("max")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, max_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, max_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, max_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, max_rr (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t1)
      end select
   case ("min")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, min_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, min_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, min_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, min_rr (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t1)
      end select
   case ("mod")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, mod_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, mod_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
```

```
                end select
            case (V_REAL)
               select case (t2)
               case (V_INT);  call eval_node_init_real (en, mod_ri (en1, en2))
               case (V_REAL); call eval_node_init_real (en, mod_rr (en1, en2))
               case default;  call eval_type_error (pn, char (key), t2)
               end select
             case default;  call eval_type_error (pn, char (key), t1)
            end select
         case ("modulo")
            select case (t1)
            case (V_INT)
               select case (t2)
               case (V_INT);  call eval_node_init_int  (en, modulo_ii (en1, en2))
               case (V_REAL); call eval_node_init_real (en, modulo_ir (en1, en2))
               case default;  call eval_type_error (pn, char (key), t2)
               end select
            case (V_REAL)
               select case (t2)
               case (V_INT);  call eval_node_init_real (en, modulo_ri (en1, en2))
               case (V_REAL); call eval_node_init_real (en, modulo_rr (en1, en2))
               case default;  call eval_type_error (pn, char (key), t2)
               end select
             case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default
            call parse_node_mismatch ("function name", pn_fname)
         end select
         call eval_node_final_rec (en1)
         deallocate (en1)
      else
         call eval_node_init_branch (en, key, t1, en1, en2)
         select case (char (key))
         case ("max")
            select case (t1)
            case (V_INT)
               select case (t2)
               case (V_INT);  call eval_node_set_op2_int  (en, max_ii)
               case (V_REAL); call eval_node_set_op2_real (en, max_ir)
               case default;  call eval_type_error (pn, char (key), t2)
               end select
            case (V_REAL)
               select case (t2)
               case (V_INT);  call eval_node_set_op2_real (en, max_ri)
               case (V_REAL); call eval_node_set_op2_real (en, max_rr)
               case default;  call eval_type_error (pn, char (key), t2)
               end select
             case default;  call eval_type_error (pn, char (key), t2)
            end select
         case ("min")
            select case (t1)
            case (V_INT)
               select case (t2)
               case (V_INT);  call eval_node_set_op2_int  (en, min_ii)
```

```
             case (V_REAL); call eval_node_set_op2_real (en, min_ir)
             case default;  call eval_type_error (pn, char (key), t2)
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_set_op2_real (en, min_ri)
             case (V_REAL); call eval_node_set_op2_real (en, min_rr)
             case default;  call eval_type_error (pn, char (key), t2)
             end select
           case default;  call eval_type_error (pn, char (key), t2)
        end select
     case ("mod")
        select case (t1)
        case (V_INT)
           select case (t2)
           case (V_INT);  call eval_node_set_op2_int  (en, mod_ii)
           case (V_REAL); call eval_node_set_op2_real (en, mod_ir)
           case default;  call eval_type_error (pn, char (key), t2)
           end select
        case (V_REAL)
           select case (t2)
           case (V_INT);  call eval_node_set_op2_real (en, mod_ri)
           case (V_REAL); call eval_node_set_op2_real (en, mod_rr)
           case default;  call eval_type_error (pn, char (key), t2)
           end select
         case default;  call eval_type_error (pn, char (key), t2)
       end select
     case ("modulo")
        select case (t1)
        case (V_INT)
           select case (t2)
           case (V_INT);  call eval_node_set_op2_int  (en, modulo_ii)
           case (V_REAL); call eval_node_set_op2_real (en, modulo_ir)
           case default;  call eval_type_error (pn, char (key), t2)
           end select
        case (V_REAL)
           select case (t2)
           case (V_INT);  call eval_node_set_op2_real (en, modulo_ri)
           case (V_REAL); call eval_node_set_op2_real (en, modulo_rr)
           case default;  call eval_type_error (pn, char (key), t2)
           end select
         case default;  call eval_type_error (pn, char (key), t2)
       end select
     case default
        call parse_node_mismatch ("function name", pn_fname)
     end select
  end if
  if (debug) then
     call eval_node_write (en)
     print *, "done function"
  end if
end subroutine eval_node_compile_binary_function
```

Variable definition

A block expression contains a variable definition (first argument) and an expression where the definition can be used (second argument). The `result_type` decides which type of expression is expected for the second argument. For numeric variables, if there is a mismatch between real and integer type, insert an extra node for type conversion.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_block_expr &
    (en, pn, var_list, result_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: result_type
  type(parse_node_t), pointer :: pn_var_spec, pn_var_subspec
  type(parse_node_t), pointer :: pn_var_type, pn_var_name, pn_var_expr
  type(parse_node_t), pointer :: pn_expr
  type(string_t) :: var_name
  type(eval_node_t), pointer :: en1, en2
  integer :: var_type
  logical :: new
  if (debug) then
     print *, "read block expr";  call parse_node_write (pn)
  end if
  new = .false.
  pn_var_spec => parse_node_get_sub_ptr (pn, 2)
  select case (char (parse_node_get_rule_key (pn_var_spec)))
  case ("var_num");       var_type = V_NONE
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec)
  case ("var_int");       var_type = V_INT
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_real");      var_type = V_REAL
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_cmplx");     var_type = V_CMPLX
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_logical_new");  var_type = V_LOG
     new = .true.
     pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
     pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
  case ("var_logical_spec");  var_type = V_LOG
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_plist_new");    var_type = V_SEV
     new = .true.
     pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
     pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
  case ("var_plist_spec");    var_type = V_SEV
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_alias");     var_type = V_PDG
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
```

```
      case ("var_string_new");    var_type = V_STR
         new = .true.
         pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
         pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
      case ("var_string_spec");    var_type = V_STR
         pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
      case default
         call parse_node_mismatch &
               ("logical|int|real|plist|alias", pn_var_type)
      end select
      pn_var_expr => parse_node_get_next_ptr (pn_var_name, 2)
      pn_expr => parse_node_get_next_ptr (pn_var_spec, 2)
      var_name = parse_node_get_string (pn_var_name)
      select case (var_type)
      case (V_LOG);   var_name = "?" // var_name
      case (V_SEV);   var_name = "@" // var_name
      case (V_STR);   var_name = "$" // var_name     ! $ sign
      end select
      call var_list_check_user_var (var_list, var_name, var_type, new)
      call eval_node_compile_genexpr (en1, pn_var_expr, var_list, var_type)
      call insert_conversion_node (en1, var_type)
      allocate (en)
      call eval_node_init_block (en, var_name, var_type, en1, var_list)
      call eval_node_compile_genexpr (en2, pn_expr, en%var_list, result_type)
      call eval_node_set_expr (en, en2)
      if (debug) then
         call eval_node_write (en)
         print *, "done block expr"
      end if
   end subroutine eval_node_compile_block_expr
```

Insert a conversion node for integer/real/complex transformation if necessary.
What shall we do for the complex to integer/real conversion?

⟨*Expressions: procedures*⟩+≡

```
   subroutine insert_conversion_node (en, result_type)
     type(eval_node_t), pointer :: en
     integer, intent(in) :: result_type
     type(eval_node_t), pointer :: en_conv
     select case (en%result_type)
     case (V_INT)
        select case (result_type)
        case (V_REAL)
           allocate (en_conv)
           call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
           call eval_node_set_op1_real (en_conv, real_i)
           en => en_conv
        case (V_CMPLX)
           allocate (en_conv)
           call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
           call eval_node_set_op1_cmplx (en_conv, cmplx_i)
           en => en_conv
        end select
     case (V_REAL)
```

```
      select case (result_type)
      case (V_INT)
         allocate (en_conv)
         call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
         call eval_node_set_op1_int (en_conv, int_r)
         en => en_conv
      case (V_CMPLX)
         allocate (en_conv)
         call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
         call eval_node_set_op1_cmplx (en_conv, cmplx_r)
         en => en_conv
      end select
   case (V_CMPLX)
      select case (result_type)
      case (V_INT)
         allocate (en_conv)
         call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
         call eval_node_set_op1_int (en_conv, int_c)
         en => en_conv
      case (V_REAL)
         allocate (en_conv)
         call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
         call eval_node_set_op1_real (en_conv, real_c)
         en => en_conv
      end select
    case default
    end select
  end subroutine insert_conversion_node
```

Conditionals

A conditional has the structure if lexpr then expr else expr. So we first evaluate the logical expression, then depending on the result the first or second expression. Note that the second expression is mandatory.

The `result_type`, if present, defines the requested type of the **then** and **else** clauses. Default is numeric (int/real). If there is a mismatch between real and integer result types, insert conversion nodes.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_conditional &
        (en, pn, var_list, result_type)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     integer, intent(in), optional :: result_type
     type(parse_node_t), pointer :: pn_condition, pn_expr
     type(parse_node_t), pointer :: pn_maybe_elsif, pn_elsif_branch
     type(parse_node_t), pointer :: pn_maybe_else, pn_else_branch, pn_else_expr
     type(eval_node_t), pointer :: en0, en1, en2
     integer :: restype
     if (debug) then
        print *, "read conditional";  call parse_node_write (pn)
     end if
```

```
pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
pn_expr => parse_node_get_next_ptr (pn_condition, 2)
call eval_node_compile_lexpr (en0, pn_condition, var_list)
call eval_node_compile_genexpr (en1, pn_expr, var_list, result_type)
if (present (result_type)) then
   restype = major_result_type (result_type, en1%result_type)
else
   restype = en1%result_type
end if
pn_maybe_elsif => parse_node_get_next_ptr (pn_expr)
select case (char (parse_node_get_rule_key (pn_maybe_elsif)))
case ("maybe_elsif_expr", &
      "maybe_elsif_lexpr", &
      "maybe_elsif_pexpr", &
      "maybe_elsif_cexpr", &
      "maybe_elsif_sexpr")
   pn_elsif_branch => parse_node_get_sub_ptr (pn_maybe_elsif)
   pn_maybe_else => parse_node_get_next_ptr (pn_maybe_elsif)
   select case (char (parse_node_get_rule_key (pn_maybe_else)))
   case ("maybe_else_expr", &
      "maybe_else_lexpr", &
      "maybe_else_pexpr", &
      "maybe_else_cexpr", &
      "maybe_else_sexpr")
      pn_else_branch => parse_node_get_sub_ptr (pn_maybe_else)
      pn_else_expr => parse_node_get_sub_ptr (pn_else_branch, 2)
   case default
      pn_else_expr => null ()
   end select
   call eval_node_compile_elsif &
        (en2, pn_elsif_branch, pn_else_expr, var_list, restype)
case ("maybe_else_expr", &
      "maybe_else_lexpr", &
      "maybe_else_pexpr", &
      "maybe_else_cexpr", &
      "maybe_else_sexpr")
   pn_maybe_else => pn_maybe_elsif
   pn_maybe_elsif => null ()
   pn_else_branch => parse_node_get_sub_ptr (pn_maybe_else)
   pn_else_expr => parse_node_get_sub_ptr (pn_else_branch, 2)
   call eval_node_compile_genexpr &
        (en2, pn_else_expr, var_list, restype)
case ("endif")
   call eval_node_compile_default_else (en2, restype)
case default
   call msg_bug ("Broken conditional: unexpected " &
        // char (parse_node_get_rule_key (pn_maybe_elsif)))
end select
call eval_node_create_conditional (en, en0, en1, en2, restype)
call conditional_insert_conversion_nodes (en, restype)
if (debug) then
   call eval_node_write (en)
   print *, "done conditional"
end if
```

```
        end subroutine eval_node_compile_conditional
```

This recursively generates 'elsif' conditionals as a chain of sub-nodes of the main
conditional.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_elsif &
         (en, pn, pn_else_expr, var_list, result_type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(parse_node_t), pointer :: pn_else_expr
      type(var_list_t), intent(in), target :: var_list
      integer, intent(inout) :: result_type
      type(parse_node_t), pointer :: pn_next, pn_condition, pn_expr
      type(eval_node_t), pointer :: en0, en1, en2
      pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
      pn_expr => parse_node_get_next_ptr (pn_condition, 2)
      call eval_node_compile_lexpr (en0, pn_condition, var_list)
      call eval_node_compile_genexpr (en1, pn_expr, var_list, result_type)
      result_type = major_result_type (result_type, en1%result_type)
      pn_next => parse_node_get_next_ptr (pn)
      if (associated (pn_next)) then
         call eval_node_compile_elsif &
              (en2, pn_next, pn_else_expr, var_list, result_type)
         result_type = major_result_type (result_type, en2%result_type)
      else if (associated (pn_else_expr)) then
         call eval_node_compile_genexpr &
              (en2, pn_else_expr, var_list, result_type)
         result_type = major_result_type (result_type, en2%result_type)
      else
         call eval_node_compile_default_else (en2, result_type)
      end if
      call eval_node_create_conditional (en, en0, en1, en2, result_type)
    end subroutine eval_node_compile_elsif
```

This makes a default 'else' branch in case it was omitted. The default value just
depends on the expected type.

⟨*Expressions: procedures*⟩+≡

```
    subroutine eval_node_compile_default_else (en, result_type)
      type(eval_node_t), pointer :: en
      integer, intent(in) :: result_type
      type(subevt_t) :: pval_empty
      type(pdg_array_t) :: aval_undefined
      allocate (en)
      select case (result_type)
      case (V_LOG);  call eval_node_init_log (en, .false.)
      case (V_INT);  call eval_node_init_int (en, 0)
      case (V_REAL);  call eval_node_init_real (en, 0._default)
      case (V_CMPLX)
           call eval_node_init_cmplx (en, (0._default, 0._default))
      case (V_SEV)
         call subevt_init (pval_empty)
         call eval_node_init_subevt (en, pval_empty)
      case (V_PDG)
```

```
         call eval_node_init_pdg_array  (en, aval_undefined)
      case (V_STR)
         call eval_node_init_string (en, var_str (""))
      case default
         call msg_bug ("Undefined type for 'else' branch in conditional")
      end select
   end subroutine eval_node_compile_default_else
```

If the logical expression is constant, we can simplify the conditional node by
replacing it with the selected branch. Otherwise, we initialize a true branching.

⟨*Expressions: procedures*⟩+≡

```
   subroutine eval_node_create_conditional (en, en0, en1, en2, result_type)
     type(eval_node_t), pointer :: en, en0, en1, en2
     integer, intent(in) :: result_type
     if (en0%type == EN_CONSTANT) then
        if (en0%lval) then
           en => en1
           call eval_node_final_rec (en2)
           deallocate (en2)
        else
           en => en2
           call eval_node_final_rec (en1)
           deallocate (en1)
        end if
     else
        allocate (en)
        call eval_node_init_conditional (en, result_type, en0, en1, en2)
     end if
   end subroutine eval_node_create_conditional
```

Return the numerical result type which should be used for the combination of
the two result types.

⟨*Expressions: procedures*⟩+≡

```
   function major_result_type (t1, t2) result (t)
     integer :: t
     integer, intent(in) :: t1, t2
     select case (t1)
     case (V_INT)
        select case (t2)
        case (V_INT, V_REAL, V_CMPLX)
           t = t2
        case default
           call type_mismatch ()
        end select
     case (V_REAL)
        select case (t2)
        case (V_INT)
           t = t1
        case (V_REAL, V_CMPLX)
           t = t2
        case default
           call type_mismatch ()
        end select
```

```
      case (V_CMPLX)
         select case (t2)
         case (V_INT, V_REAL, V_CMPLX)
            t = t1
         case default
            call type_mismatch ()
         end select
      case default
         if (t1 == t2) then
            t = t1
         else
            call type_mismatch ()
         end if
      end select
   contains
      subroutine type_mismatch ()
         call msg_bug ("Type mismatch in branches of a conditional expression")
      end subroutine type_mismatch
   end function major_result_type
```

Recursively insert conversion nodes where necessary.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine conditional_insert_conversion_nodes (en, result_type)
      type(eval_node_t), intent(inout), target :: en
      integer, intent(in) :: result_type
      select case (result_type)
      case (V_INT, V_REAL, V_CMPLX)
         call insert_conversion_node (en%arg1, result_type)
         if (en%arg2%type == EN_CONDITIONAL) then
            call conditional_insert_conversion_nodes (en%arg2, result_type)
         else
            call insert_conversion_node (en%arg2, result_type)
         end if
      end select
   end subroutine conditional_insert_conversion_nodes
```

Logical expressions

A logical expression consists of one or more singlet logical expressions concatenated by ;. This is for allowing side-effects, only the last value is used.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_lexpr (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_term, pn_sequel, pn_arg
      type(eval_node_t), pointer :: en1, en2
      if (debug) then
         print *, "read lexpr";  call parse_node_write (pn)
      end if
      pn_term => parse_node_get_sub_ptr (pn, tag="lsinglet")
      call eval_node_compile_lsinglet (en, pn_term, var_list)
```

```
      pn_sequel => parse_node_get_next_ptr (pn_term, tag="lsequel")
      do while (associated (pn_sequel))
         pn_arg => parse_node_get_sub_ptr (pn_sequel, 2, tag="lsinglet")
         en1 => en
         call eval_node_compile_lsinglet (en2, pn_arg, var_list)
         allocate (en)
         if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call eval_node_init_log (en, ignore_first_ll (en1, en2))
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
         else
            call eval_node_init_branch &
                 (en, var_str ("lsequel"), V_LOG, en1, en2)
            call eval_node_set_op2_log (en, ignore_first_ll)
         end if
         pn_sequel => parse_node_get_next_ptr (pn_sequel)
      end do
      if (debug) then
         call eval_node_write (en)
         print *, "done lexpr"
      end if
   end subroutine eval_node_compile_lexpr
```

A logical singlet expression consists of one or more logical terms concatenated
by or.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_lsinglet (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_term, pn_alternative, pn_arg
     type(eval_node_t), pointer :: en1, en2
     if (debug) then
        print *, "read lsinglet";  call parse_node_write (pn)
     end if
     pn_term => parse_node_get_sub_ptr (pn, tag="lterm")
     call eval_node_compile_lterm (en, pn_term, var_list)
     pn_alternative => parse_node_get_next_ptr (pn_term, tag="alternative")
     do while (associated (pn_alternative))
        pn_arg => parse_node_get_sub_ptr (pn_alternative, 2, tag="lterm")
        en1 => en
        call eval_node_compile_lterm (en2, pn_arg, var_list)
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
           call eval_node_init_log (en, or_ll (en1, en2))
           call eval_node_final_rec (en1)
           call eval_node_final_rec (en2)
           deallocate (en1, en2)
        else
           call eval_node_init_branch &
                (en, var_str ("alternative"), V_LOG, en1, en2)
           call eval_node_set_op2_log (en, or_ll)
```

```
      end if
      pn_alternative => parse_node_get_next_ptr (pn_alternative)
   end do
   if (debug) then
      call eval_node_write (en)
      print *, "done lsinglet"
   end if
end subroutine eval_node_compile_lsinglet
```

A logical term consists of one or more logical values concatenated by **and**.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_lterm (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_term, pn_coincidence, pn_arg
     type(eval_node_t), pointer :: en1, en2
     if (debug) then
        print *, "read lterm";  call parse_node_write (pn)
     end if
     pn_term => parse_node_get_sub_ptr (pn)
     call eval_node_compile_lvalue (en, pn_term, var_list)
     pn_coincidence => parse_node_get_next_ptr (pn_term, tag="coincidence")
     do while (associated (pn_coincidence))
        pn_arg => parse_node_get_sub_ptr (pn_coincidence, 2)
        en1 => en
        call eval_node_compile_lvalue (en2, pn_arg, var_list)
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
           call eval_node_init_log (en, and_ll (en1, en2))
           call eval_node_final_rec (en1)
           call eval_node_final_rec (en2)
           deallocate (en1, en2)
        else
           call eval_node_init_branch &
                (en, var_str ("coincidence"), V_LOG, en1, en2)
           call eval_node_set_op2_log (en, and_ll)
        end if
        pn_coincidence => parse_node_get_next_ptr (pn_coincidence)
     end do
     if (debug) then
        call eval_node_write (en)
        print *, "done lterm"
     end if
   end subroutine eval_node_compile_lterm
```

Logical variables are disabled, because they are confused with the l.h.s. of compared expressions.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_lvalue (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
```

```
      if (debug) then
         print *, "read lvalue";  call parse_node_write (pn)
      end if
      select case (char (parse_node_get_rule_key (pn)))
      case ("true")
         allocate (en)
         call eval_node_init_log (en, .true.)
      case ("false")
         allocate (en)
         call eval_node_init_log (en, .false.)
      case ("negation")
         call eval_node_compile_negation (en, pn, var_list)
      case ("lvariable")
         call eval_node_compile_variable (en, pn, var_list, V_LOG)
      case ("lexpr")
         call eval_node_compile_lexpr (en, pn, var_list)
      case ("block_lexpr")
         call eval_node_compile_block_expr (en, pn, var_list, V_LOG)
      case ("conditional_lexpr")
         call eval_node_compile_conditional (en, pn, var_list, V_LOG)
      case ("compared_expr")
         call eval_node_compile_compared_expr (en, pn, var_list, V_REAL)
      case ("compared_sexpr")
         call eval_node_compile_compared_expr (en, pn, var_list, V_STR)
      case ("all_fun", "any_fun", "no_fun", "user_cut_fun")
         call eval_node_compile_log_function (en, pn, var_list)
      case ("record_cmd")
         call eval_node_compile_record_cmd (en, pn, var_list)
      case default
         call parse_node_mismatch &
              ("true|false|negation|lvariable|" // &
               "lexpr|block_lexpr|conditional_lexpr|" // &
               "compared_expr|compared_sexpr|logical_pexpr", pn)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done lvalue"
      end if
    end subroutine eval_node_compile_lvalue
```

A negation consists of the keyword **not** and a logical value.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_negation (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_arg
      type(eval_node_t), pointer :: en1
      if (debug) then
         print *, "read negation";  call parse_node_write (pn)
      end if
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      call eval_node_compile_lvalue (en1, pn_arg, var_list)
      allocate (en)
```

```
      if (en1%type == EN_CONSTANT) then
         call eval_node_init_log (en, not_l (en1))
         call eval_node_final_rec (en1)
         deallocate (en1)
      else
         call eval_node_init_branch (en, var_str ("not"), V_LOG, en1)
         call eval_node_set_op1_log (en, not_l)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done negation"
      end if
   end subroutine eval_node_compile_negation
```

Comparisons

Up to the loop, this is easy. There is always at least one comparison. This is
evaluated, and the result is the logical node **en**. If it is constant, we keep its
second sub-node as **en2**. (Thus, at the very end **en2** has to be deleted if **en** is
(still) constant.)

If there is another comparison, we first check if the first comparison was
constant. In that case, there are two possibilities: (i) it was true. Then, its
right-hand side is compared with the new right-hand side, and the result replaces
the previous one which is deleted. (ii) it was false. In this case, the result of the
whole comparison is false, and we can exit the loop without evaluating anything
else.

Now assume that the first comparison results in a valid branch, its second
sub-node kept as **en2**. We first need a copy of this, which becomes the new
left-hand side. If **en2** is constant, we make an identical constant node **en1**.
Otherwise, we make **en1** an appropriate pointer node. Next, the first branch is
saved as **en0** and we evaluate the comparison between **en1** and the a right-hand
side. If this turns out to be constant, there are again two possibilities: (i) true,
then we revert to the previous result. (ii) false, then the wh

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_compared_expr (en, pn, var_list, type)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     integer, intent(in) :: type
     type(parse_node_t), pointer :: pn_comparison, pn_expr1
     type(eval_node_t), pointer :: en0, en1, en2
     if (debug) then
        print *, "read comparison";  call parse_node_write (pn)
     end if
     select case (type)
     case (V_INT, V_REAL)
        pn_expr1 => parse_node_get_sub_ptr (pn, tag="expr")
        call eval_node_compile_expr (en1, pn_expr1, var_list)
        pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="comparison")
     case (V_STR)
        pn_expr1 => parse_node_get_sub_ptr (pn, tag="sexpr")
```

```
      call eval_node_compile_sexpr (en1, pn_expr1, var_list)
   pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="str_comparison")
end select
call eval_node_compile_comparison &
      (en, en1, en2, pn_comparison, var_list, type)
pn_comparison => parse_node_get_next_ptr (pn_comparison)
SCAN_FURTHER: do while (associated (pn_comparison))
   if (en%type == EN_CONSTANT) then
      if (en%lval) then
         en1 => en2
         call eval_node_final_rec (en);  deallocate (en)
         call eval_node_compile_comparison &
               (en, en1, en2, pn_comparison, var_list, type)
      else
         exit SCAN_FURTHER
      end if
   else
      allocate (en1)
      if (en2%type == EN_CONSTANT) then
         select case (en2%result_type)
         case (V_INT);  call eval_node_init_int    (en1, en2%ival)
         case (V_REAL); call eval_node_init_real   (en1, en2%rval)
         case (V_STR);  call eval_node_init_string (en1, en2%sval)
         end select
      else
         select case (en2%result_type)
         case (V_INT);  call eval_node_init_int_ptr &
               (en1, var_str ("(previous)"), en2%ival, en2%value_is_known)
         case (V_REAL); call eval_node_init_real_ptr &
               (en1, var_str ("(previous)"), en2%rval, en2%value_is_known)
         case (V_STR);  call eval_node_init_string_ptr &
               (en1, var_str ("(previous)"), en2%sval, en2%value_is_known)
         end select
      end if
      en0 => en
      call eval_node_compile_comparison &
            (en, en1, en2, pn_comparison, var_list, type)
      if (en%type == EN_CONSTANT) then
         if (en%lval) then
            call eval_node_final_rec (en);  deallocate (en)
            en => en0
         else
            call eval_node_final_rec (en0);  deallocate (en0)
            exit SCAN_FURTHER
         end if
      else
         en1 => en
         allocate (en)
         call eval_node_init_branch (en, var_str ("and"), V_LOG, en0, en1)
         call eval_node_set_op2_log (en, and_ll)
      end if
   end if
   pn_comparison => parse_node_get_next_ptr (pn_comparison)
end do SCAN_FURTHER
```

```
      if (en%type == EN_CONSTANT .and. associated (en2)) then
         call eval_node_final_rec (en2);  deallocate (en2)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done compared_expr"
      end if
   end subroutine eval_node_compile_compared_expr
```

This takes two extra arguments: **en1**, the left-hand-side of the comparison, is already allocated and evaluated. **en2** (the right-hand side) and **en** (the result) are allocated by the routine. **pn** is the parse node which contains the operator and the right-hand side as subnodes.

If the result of the comparison is constant, **en1** is deleted but **en2** is kept, because it may be used in a subsequent comparison. **en** then becomes a constant. If the result is variable, **en** becomes a branch node which refers to **en1** and **en2**.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_compile_comparison &
        (en, en1, en2, pn, var_list, type)
     type(eval_node_t), pointer :: en, en1, en2
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     integer, intent(in) :: type
     type(parse_node_t), pointer :: pn_op, pn_arg
     type(string_t) :: key
     integer :: t1, t2
     type(var_entry_t), pointer :: var
     pn_op => parse_node_get_sub_ptr (pn)
     key = parse_node_get_key (pn_op)
     select case (type)
     case (V_INT, V_REAL)
        pn_arg => parse_node_get_next_ptr (pn_op, tag="expr")
        call eval_node_compile_expr (en2, pn_arg, var_list)
     case (V_STR)
        pn_arg => parse_node_get_next_ptr (pn_op, tag="sexpr")
        call eval_node_compile_sexpr (en2, pn_arg, var_list)
     end select
     t1 = en1%result_type
     t2 = en2%result_type
     allocate (en)
     if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
        var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
        en1%tolerance => var_entry_get_rval_ptr (var)
        select case (char (key))
        case ("<")
           select case (t1)
           case (V_INT)
              select case (t2)
              case (V_INT);  call eval_node_init_log (en, comp_lt_ii (en1, en2))
              case (V_REAL); call eval_node_init_log (en, comp_ll_ir (en1, en2))
              end select
           case (V_REAL)
              select case (t2)
```

557

```
                case (V_INT);  call eval_node_init_log (en, comp_ll_ri (en1, en2))
                case (V_REAL); call eval_node_init_log (en, comp_ll_rr (en1, en2))
                end select
             end select
       case (">")
          select case (t1)
          case (V_INT)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_gt_ii (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_gg_ir (en1, en2))
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_gg_ri (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_gg_rr (en1, en2))
             end select
          end select
       case ("<=")
          select case (t1)
          case (V_INT)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_le_ii (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_ls_ir (en1, en2))
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_ls_ri (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_ls_rr (en1, en2))
             end select
          end select
       case (">=")
          select case (t1)
          case (V_INT)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_ge_ii (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_gs_ir (en1, en2))
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_gs_ri (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_gs_rr (en1, en2))
             end select
          end select
       case ("==")
          select case (t1)
          case (V_INT)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_eq_ii (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_se_ir (en1, en2))
             end select
          case (V_REAL)
             select case (t2)
             case (V_INT);  call eval_node_init_log (en, comp_se_ri (en1, en2))
             case (V_REAL); call eval_node_init_log (en, comp_se_rr (en1, en2))
```

```
                     end select
               case (V_STR)
                  select case (t2)
                  case (V_STR);  call eval_node_init_log (en, comp_eq_ss (en1, en2))
                  end select
               end select
            case ("<>")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);  call eval_node_init_log (en, comp_ne_ii (en1, en2))
                  case (V_REAL); call eval_node_init_log (en, comp_ns_ir (en1, en2))
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);  call eval_node_init_log (en, comp_ns_ri (en1, en2))
                  case (V_REAL); call eval_node_init_log (en, comp_ns_rr (en1, en2))
                  end select
               case (V_STR)
                  select case (t2)
                  case (V_STR);  call eval_node_init_log (en, comp_ne_ss (en1, en2))
                  end select
               end select
            end select
            call eval_node_final_rec (en1)
            deallocate (en1)
         else
            call eval_node_init_branch (en, key, V_LOG, en1, en2)
            select case (char (key))
            case ("<")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_lt_ii)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_ll_ir)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_ll_ri)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_ll_rr)
                  end select
               end select
            case (">")
               select case (t1)
               case (V_INT)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_gt_ii)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_gg_ir)
                  end select
               case (V_REAL)
                  select case (t2)
                  case (V_INT);  call eval_node_set_op2_log (en, comp_gg_ri)
                  case (V_REAL); call eval_node_set_op2_log (en, comp_gg_rr)
                  end select
```

```
      end select
case ("<=")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_le_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ls_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ls_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ls_rr)
      end select
   end select
case (">=")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ge_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_gs_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_gs_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_gs_rr)
      end select
   end select
case ("==")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_eq_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_se_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_se_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_se_rr)
      end select
   case (V_STR)
      select case (t2)
      case (V_STR);  call eval_node_set_op2_log (en, comp_eq_ss)
      end select
   end select
case ("<>")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ne_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ns_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ns_ri)
```

```
            case (V_REAL); call eval_node_set_op2_log (en, comp_ns_rr)
            end select
         case (V_STR)
            select case (t2)
            case (V_STR);  call eval_node_set_op2_log (en, comp_ne_ss)
            end select
         end select
      end select
      var => var_list_get_var_ptr (var_list, var_str ("tolerance"))
      en1%tolerance => var_entry_get_rval_ptr (var)
   end if
 end subroutine eval_node_compile_comparison
```

Recording analysis data

The `record` command is actually a logical expression which always evaluates
`true`.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_record_cmd (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_key, pn_tag, pn_arg
     type(parse_node_t), pointer :: pn_arg1, pn_arg2, pn_arg3, pn_arg4
     type(eval_node_t), pointer :: en0, en1, en2, en3, en4
     type(var_entry_t), pointer :: var
     real(default), pointer :: event_weight
     if (debug) then
        print *, "read record_cmd";  call parse_node_write (pn)
     end if
     pn_key => parse_node_get_sub_ptr (pn)
     pn_tag => parse_node_get_next_ptr (pn_key)
     pn_arg => parse_node_get_next_ptr (pn_tag)
     select case (char (parse_node_get_key (pn_key)))
     case ("record")
        var => var_list_get_var_ptr (var_list, var_str ("event_weight"))
        if (associated (var)) then
           event_weight => var_entry_get_rval_ptr (var)
        else
           event_weight => null ()
        end if
     case ("record_unweighted")
        event_weight => null ()
     case ("record_excess")
        var => var_list_get_var_ptr (var_list, var_str ("event_excess"))
        if (associated (var)) then
           event_weight => var_entry_get_rval_ptr (var)
        else
           event_weight => null ()
        end if
     end select
     select case (char (parse_node_get_rule_key (pn_tag)))
```

```
      case ("analysis_id")
         allocate (en0)
         call eval_node_init_string (en0, parse_node_get_string (pn_tag))
      case default
         call eval_node_compile_sexpr (en0, pn_tag, var_list)
      end select
      allocate (en)
      if (associated (pn_arg)) then
         pn_arg1 => parse_node_get_sub_ptr (pn_arg)
         call eval_node_compile_expr (en1, pn_arg1, var_list)
         if (en1%result_type == V_INT) &
              call insert_conversion_node (en1, V_REAL)
         pn_arg2 => parse_node_get_next_ptr (pn_arg1)
         if (associated (pn_arg2)) then
            call eval_node_compile_expr (en2, pn_arg2, var_list)
            if (en2%result_type == V_INT) &
                 call insert_conversion_node (en2, V_REAL)
            pn_arg3 => parse_node_get_next_ptr (pn_arg2)
            if (associated (pn_arg3)) then
               call eval_node_compile_expr (en3, pn_arg3, var_list)
               if (en3%result_type == V_INT) &
                    call insert_conversion_node (en3, V_REAL)
               pn_arg4 => parse_node_get_next_ptr (pn_arg3)
               if (associated (pn_arg4)) then
                  call eval_node_compile_expr (en4, pn_arg4, var_list)
                  if (en4%result_type == V_INT) &
                       call insert_conversion_node (en4, V_REAL)
                  call eval_node_init_record_cmd &
                       (en, event_weight, en0, en1, en2, en3, en4)
               else
                  call eval_node_init_record_cmd &
                       (en, event_weight, en0, en1, en2, en3)
               end if
            else
               call eval_node_init_record_cmd (en, event_weight, en0, en1, en2)
            end if
         else
            call eval_node_init_record_cmd (en, event_weight, en0, en1)
         end if
      else
         call eval_node_init_record_cmd (en, event_weight, en0)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done record_cmd"
      end if
   end subroutine eval_node_compile_record_cmd
```

Particle-list expressions

A particle expression is a subevent or a concatenation of particle-list terms
(using `join`).
⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_pexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_pterm, pn_concatenation, pn_op, pn_arg
  type(eval_node_t), pointer :: en1, en2
  type(subevt_t) :: subevt
  if (debug) then
     print *, "read pexpr";  call parse_node_write (pn)
  end if
  pn_pterm => parse_node_get_sub_ptr (pn)
  call eval_node_compile_pterm (en, pn_pterm, var_list)
  pn_concatenation => &
       parse_node_get_next_ptr (pn_pterm, tag="pconcatenation")
  do while (associated (pn_concatenation))
     pn_op => parse_node_get_sub_ptr (pn_concatenation)
     pn_arg => parse_node_get_next_ptr (pn_op)
     en1 => en
     call eval_node_compile_pterm (en2, pn_arg, var_list)
     allocate (en)
     if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
        call subevt_join (subevt, en1%pval, en2%pval)
        call eval_node_init_subevt (en, subevt)
        call eval_node_final_rec (en1)
        call eval_node_final_rec (en2)
        deallocate (en1, en2)
     else
        call eval_node_init_branch &
             (en, var_str ("join"), V_SEV, en1, en2)
        call eval_node_set_op2_sev (en, join_pp)
     end if
     pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
  end do
  if (debug) then
     call eval_node_write (en)
     print *, "done pexpr"
  end if
end subroutine eval_node_compile_pexpr
```

A particle term is a subevent or a combination of particle-list values (using
combine).

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_pterm (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_pvalue, pn_combination, pn_op, pn_arg
  type(eval_node_t), pointer :: en1, en2
  type(subevt_t) :: subevt
  if (debug) then
     print *, "read pterm";  call parse_node_write (pn)
  end if
  pn_pvalue => parse_node_get_sub_ptr (pn)
```

```
        call eval_node_compile_pvalue (en, pn_pvalue, var_list)
    pn_combination => &
          parse_node_get_next_ptr (pn_pvalue, tag="pcombination")
    do while (associated (pn_combination))
        pn_op => parse_node_get_sub_ptr (pn_combination)
        pn_arg => parse_node_get_next_ptr (pn_op)
        en1 => en
        call eval_node_compile_pvalue (en2, pn_arg, var_list)
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call subevt_combine (subevt, en1%pval, en2%pval)
            call eval_node_init_subevt (en, subevt)
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
        else
            call eval_node_init_branch &
                  (en, var_str ("combine"), V_SEV, en1, en2)
            call eval_node_set_op2_sev (en, combine_pp)
        end if
        pn_combination => parse_node_get_next_ptr (pn_combination)
    end do
    if (debug) then
        call eval_node_write (en)
        print *, "done pterm"
    end if
  end subroutine eval_node_compile_pterm
```

A particle-list value is a PDG-code array, a particle identifier, a variable, a
(grouped) pexpr, a block pexpr, a conditional, or a particle-list function.

The `cexpr` node is responsible for transforming a constant PDG-code array
into a subevent. It takes the code array as its first argument, the event subevent
as its second argument, and the requested particle type (incoming/outgoing) as
its zero-th argument. The result is the list of particles in the event that match
the code array.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_pvalue (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_prefix_cexpr
      type(eval_node_t), pointer :: en1, en2, en0
      type(string_t) :: key
      type(var_entry_t), pointer :: var
      logical, save, target :: known = .true.
      if (debug) then
         print *, "read pvalue";  call parse_node_write (pn)
      end if
      select case (char (parse_node_get_rule_key (pn)))
      case ("pexpr_src")
         call eval_node_compile_prefix_cexpr (en1, pn, var_list)
         allocate (en2)
         var => var_list_get_var_ptr (var_list, var_str ("@evt"))
```

```
              if (associated (var)) then
                  call eval_node_init_subevt_ptr &
                      (en2, var_str ("@evt"), var_entry_get_pval_ptr (var), known)
                  allocate (en)
                  call eval_node_init_branch &
                      (en, var_str ("prt_selection"), V_SEV, en1, en2)
                  call eval_node_set_op2_sev (en, select_pdg_ca)
                  allocate (en0)
                  pn_prefix_cexpr => parse_node_get_sub_ptr (pn)
                  key = parse_node_get_rule_key (pn_prefix_cexpr)
                  select case (char (key))
                  case ("incoming_prt")
                     call eval_node_init_int (en0, PRT_INCOMING)
                     en%arg0 => en0
                  case ("outgoing_prt")
                     call eval_node_init_int (en0, PRT_OUTGOING)
                     en%arg0 => en0
                  end select
              else
                  call parse_node_write (pn)
                  call msg_bug (" Missing event data while compiling pvalue")
              end if
          case ("pvariable")
              call eval_node_compile_variable (en, pn, var_list, V_SEV)
          case ("pexpr")
              call eval_node_compile_pexpr (en, pn, var_list)
          case ("block_pexpr")
              call eval_node_compile_block_expr (en, pn, var_list, V_SEV)
          case ("conditional_pexpr")
              call eval_node_compile_conditional (en, pn, var_list, V_SEV)
          case ("join_fun", "combine_fun", "collect_fun", "cluster_fun", &
                  "select_fun", "extract_fun", "sort_fun")
              call eval_node_compile_prt_function (en, pn, var_list)
          case default
              call parse_node_mismatch &
                  ("prefix_cexpr|pvariable|" // &
                   "grouped_pexpr|block_pexpr|conditional_pexpr|" // &
                   "prt_function", pn)
          end select
          if (debug) then
              call eval_node_write (en)
              print *, "done pvalue"
          end if
      end subroutine eval_node_compile_pvalue
```

Particle functions

This combines the treatment of 'join', 'combine', 'collect', 'cluster', 'select', and 'extract' which all have the same syntax. The one or two argument nodes are allocated. If there is a condition, the condition node is also allocated as a logical expression, for which the variable list is augmented by the appropriate (unary/binary) observables.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_prt_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
      type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
      type(eval_node_t), pointer :: en0, en1, en2
      type(string_t) :: key
      type(var_entry_t), pointer :: var
      if (debug) then
         print *, "read prt_function";  call parse_node_write (pn)
      end if
      pn_clause => parse_node_get_sub_ptr (pn)
      pn_key  => parse_node_get_sub_ptr (pn_clause)
      pn_cond => parse_node_get_next_ptr (pn_key)
      if (associated (pn_cond)) &
           pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
      pn_args => parse_node_get_next_ptr (pn_clause)
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      key = parse_node_get_key (pn_key)
      call eval_node_compile_pexpr (en1, pn_arg1, var_list)
      allocate (en)
      if (.not. associated (pn_arg2)) then
         select case (char (key))
         case ("collect")
            call eval_node_init_prt_fun_unary (en, en1, key, collect_p)
         case ("cluster")
            if (fastjet_available ()) then
               call fastjet_init ()
            else
               call msg_fatal &
                 ("'cluster' function requires FastJet, which is not enabled")
            end if
            call eval_node_init_prt_fun_unary (en, en1, key, cluster_p)
            var => var_list_get_var_ptr (var_list, var_str ("jet_algorithm"))
            en1%jet_algorithm => var_entry_get_ival_ptr (var)
            var => var_list_get_var_ptr (var_list, var_str ("jet_r"))
            en1%jet_r => var_entry_get_rval_ptr (var)
         case ("select")
            call eval_node_init_prt_fun_unary (en, en1, key, select_p)
         case ("extract")
            call eval_node_init_prt_fun_unary (en, en1, key, extract_p)
         case ("sort")
            call eval_node_init_prt_fun_unary (en, en1, key, sort_p)
         case default
            call msg_bug (" Unary particle function '" // char (key) // &
                 "' undefined")
         end select
      else
         call eval_node_compile_pexpr (en2, pn_arg2, var_list)
         select case (char (key))
         case ("join")
```

```
              call eval_node_init_prt_fun_binary (en, en1, en2, key, join_pp)
          case ("combine")
              call eval_node_init_prt_fun_binary (en, en1, en2, key, combine_pp)
          case ("collect")
              call eval_node_init_prt_fun_binary (en, en1, en2, key, collect_pp)
          case ("select")
              call eval_node_init_prt_fun_binary (en, en1, en2, key, select_pp)
          case ("sort")
              call eval_node_init_prt_fun_binary (en, en1, en2, key, sort_pp)
          case default
              call msg_bug (" Binary particle function '" // char (key) // &
                   "' undefined")
          end select
       end if
       if (associated (pn_cond)) then
          call eval_node_set_observables (en, var_list)
          select case (char (key))
          case ("extract", "sort")
              call eval_node_compile_expr (en0, pn_arg0, en%var_list)
          case default
              call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
          end select
          en%arg0 => en0
       end if
       if (debug) then
          call eval_node_write (en)
          print *, "done prt_function"
       end if
    end subroutine eval_node_compile_prt_function
```

The `eval` expression is similar, but here the expression `arg0` is mandatory, and the whole thing evaluates to a numeric value.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_eval_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_key, pn_arg0, pn_args, pn_arg1, pn_arg2
      type(eval_node_t), pointer :: en0, en1, en2
      type(string_t) :: key
      if (debug) then
         print *, "read eval_function";  call parse_node_write (pn)
      end if
      pn_key => parse_node_get_sub_ptr (pn)
      pn_arg0 => parse_node_get_next_ptr (pn_key)
      pn_args => parse_node_get_next_ptr (pn_arg0)
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      key = parse_node_get_key (pn_key)
      call eval_node_compile_pexpr (en1, pn_arg1, var_list)
      allocate (en)
      if (.not. associated (pn_arg2)) then
         call eval_node_init_eval_fun_unary (en, en1, key)
```

```
      else
         call eval_node_compile_pexpr (en2, pn_arg2, var_list)
         call eval_node_init_eval_fun_binary (en, en1, en2, key)
      end if
      call eval_node_set_observables (en, var_list)
      call eval_node_compile_expr (en0, pn_arg0, en%var_list)
      if (en0%result_type /= V_REAL) &
           call msg_fatal (" 'eval' function does not result in real value")
      call eval_node_set_expr (en, en0)
      if (debug) then
         call eval_node_write (en)
         print *, "done eval_function"
      end if
    end subroutine eval_node_compile_eval_function
```

Logical functions of subevents.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_log_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_key, pn_str
      type(parse_node_t), pointer :: pn_arg0, pn_args, pn_arg1, pn_arg2
      type(eval_node_t), pointer :: en0, en1, en2
      type(string_t) :: key
      if (debug) then
         print *, "read log_function";  call parse_node_write (pn)
      end if
      select case (char (parse_node_get_rule_key (pn)))
      case ("all_fun", "any_fun", "no_fun")
         pn_key => parse_node_get_sub_ptr (pn)
         pn_arg0 => parse_node_get_next_ptr (pn_key)
         pn_args => parse_node_get_next_ptr (pn_arg0)
      case ("user_cut_fun")
         pn_key => parse_node_get_sub_ptr (pn)
         pn_str => parse_node_get_next_ptr (pn_key)
         pn_arg0 => parse_node_get_sub_ptr (pn_str)
         pn_args => parse_node_get_next_ptr (pn_str)
      case default
         call parse_node_mismatch &
              ("all_fun|any_fun|no_fun|user_cut_fun", &
              pn)
      end select
      pn_arg1 => parse_node_get_sub_ptr (pn_args)
      pn_arg2 => parse_node_get_next_ptr (pn_arg1)
      key = parse_node_get_key (pn_key)
      call eval_node_compile_pexpr (en1, pn_arg1, var_list)
      allocate (en)
      if (.not. associated (pn_arg2)) then
         select case (char (key))
         case ("all")
            call eval_node_init_log_fun_unary (en, en1, key, all_p)
         case ("any")
            call eval_node_init_log_fun_unary (en, en1, key, any_p)
```

```
            case ("no")
               call eval_node_init_log_fun_unary (en, en1, key, no_p)
            case ("user_cut")
               call eval_node_init_log_fun_unary (en, en1, key, user_cut_p)
            case default
               call msg_bug ("Unary logical particle function '" // char (key) // &
                    "' undefined")
            end select
         else
            call eval_node_compile_pexpr (en2, pn_arg2, var_list)
            select case (char (key))
            case ("all")
               call eval_node_init_log_fun_binary (en, en1, en2, key, all_pp)
            case ("any")
               call eval_node_init_log_fun_binary (en, en1, en2, key, any_pp)
            case ("no")
               call eval_node_init_log_fun_binary (en, en1, en2, key, no_pp)
            case default
               call msg_bug ("Binary logical particle function '" // char (key) // &
                    "' undefined")
            end select
         end if
         if (associated (pn_arg0)) then
            call eval_node_set_observables (en, var_list)
            select case (char (key))
            case ("all", "any", "no")
               call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
            case ("user_cut")
               call eval_node_compile_sexpr (en0, pn_arg0, en%var_list)
            case default
               call msg_bug ("Compiling logical particle function: missing mode")
            end select
            call eval_node_set_expr (en, en0, V_LOG)
         end if
         if (debug) then
            call eval_node_write (en)
            print *, "done log_function"
         end if
       end subroutine eval_node_compile_log_function
```

Numeric functions of subevents.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_numeric_function (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
      type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
      type(eval_node_t), pointer :: en0, en1, en2
      type(string_t) :: key
      if (debug) then
         print *, "read numeric_function";  call parse_node_write (pn)
      end if
      select case (char (parse_node_get_rule_key (pn)))
```

569

```
case ("count_fun")
   pn_clause => parse_node_get_sub_ptr (pn)
   pn_key  => parse_node_get_sub_ptr (pn_clause)
   pn_cond => parse_node_get_next_ptr (pn_key)
   if (associated (pn_cond)) then
      pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
   else
      pn_arg0 => null ()
   end if
   pn_args => parse_node_get_next_ptr (pn_clause)
case ("user_event_fun")
   pn_key => parse_node_get_sub_ptr (pn)
   pn_cond => parse_node_get_next_ptr (pn_key)
   pn_arg0 => parse_node_get_sub_ptr (pn_cond)
   pn_args => parse_node_get_next_ptr (pn_cond)
end select
pn_arg1 => parse_node_get_sub_ptr (pn_args)
pn_arg2 => parse_node_get_next_ptr (pn_arg1)
key = parse_node_get_key (pn_key)
call eval_node_compile_pexpr (en1, pn_arg1, var_list)
allocate (en)
if (.not. associated (pn_arg2)) then
   select case (char (key))
   case ("count")
      call eval_node_init_int_fun_unary (en, en1, key, count_a)
   case ("user_event_shape")
      call eval_node_init_real_fun_unary (en, en1, key, user_event_shape_a)
   case default
      call msg_bug ("Unary subevent function '" // char (key) // &
            "' undefined")
   end select
else
   call eval_node_compile_pexpr (en2, pn_arg2, var_list)
   select case (char (key))
   case ("count")
      call eval_node_init_int_fun_binary (en, en1, en2, key, count_pp)
   case default
      call msg_bug ("Binary subevent function '" // char (key) // &
            "' undefined")
   end select
end if
if (associated (pn_arg0)) then
   call eval_node_set_observables (en, var_list)
   select case (char (key))
   case ("count")
      call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
      call eval_node_set_expr (en, en0, V_INT)
   case ("user_event_shape")
      call eval_node_compile_sexpr (en0, pn_arg0, en%var_list)
      call eval_node_set_expr (en, en0, V_REAL)
   end select
end if
if (debug) then
   call eval_node_write (en)
```

```
          print *, "done numeric_function"
       end if
    end subroutine eval_node_compile_numeric_function
```

PDG-code arrays

A PDG-code expression is either prefixed by `incoming` or `outgoing`, a block,
or a conditional. In any case, it evaluates to a constant.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_prefix_cexpr (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_avalue, pn_prt
      type(string_t) :: key
      if (debug) then
         print *, "read prefix_cexpr";  call parse_node_write (pn)
      end if
      pn_avalue => parse_node_get_sub_ptr (pn)
      key = parse_node_get_rule_key (pn_avalue)
      select case (char (key))
      case ("incoming_prt")
         pn_prt => parse_node_get_sub_ptr (pn_avalue, 2)
         call eval_node_compile_cexpr (en, pn_prt, var_list)
      case ("outgoing_prt")
         pn_prt => parse_node_get_sub_ptr (pn_avalue, 1)
         call eval_node_compile_cexpr (en, pn_prt, var_list)
      case default
         call parse_node_mismatch &
              ("incoming_prt|outgoing_prt", &
               pn_avalue)
      end select
      if (debug) then
         call eval_node_write (en)
         print *, "done prefix_cexpr"
      end if
    end subroutine eval_node_compile_prefix_cexpr
```

A PDG array is a string of PDG code definitions (or aliases), concatenated by
':'. The code definitions may be variables which are not defined at compile time,
so we have to allocate sub-nodes. This analogous to `eval_node_compile_term`.

⟨*Expressions: procedures*⟩+≡

```
    recursive subroutine eval_node_compile_cexpr (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_prt, pn_concatenation
      type(eval_node_t), pointer :: en1, en2
      type(pdg_array_t) :: aval
      if (debug) then
         print *, "read cexpr";  call parse_node_write (pn)
      end if
```

```
         pn_prt => parse_node_get_sub_ptr (pn)
         call eval_node_compile_avalue (en, pn_prt, var_list)
         pn_concatenation => parse_node_get_next_ptr (pn_prt)
         do while (associated (pn_concatenation))
            pn_prt => parse_node_get_sub_ptr (pn_concatenation, 2)
            en1 => en
            call eval_node_compile_avalue (en2, pn_prt, var_list)
            allocate (en)
            if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
               call concat_cc (aval, en1, en2)
               call eval_node_init_pdg_array (en, aval)
               call eval_node_final_rec (en1)
               call eval_node_final_rec (en2)
               deallocate (en1, en2)
            else
               call eval_node_init_branch (en, var_str (":"), V_PDG, en1, en2)
               call eval_node_set_op2_pdg (en, concat_cc)
            end if
            pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
         end do
         if (debug) then
            call eval_node_write (en)
            print *, "done cexpr"
         end if
      end subroutine eval_node_compile_cexpr
```

Compile a PDG-code type value. It may be either an integer expression or a
variable of type PDG array, optionally quoted.

⟨*Expressions: procedures*⟩+≡

```
      recursive subroutine eval_node_compile_avalue (en, pn, var_list)
        type(eval_node_t), pointer :: en
        type(parse_node_t), intent(in) :: pn
        type(var_list_t), intent(in), target :: var_list
        if (debug) then
           print *, "read avalue";  call parse_node_write (pn)
        end if
        select case (char (parse_node_get_rule_key (pn)))
        case ("pdg_code")
           call eval_node_compile_pdg_code (en, pn, var_list)
        case ("cvariable", "variable", "prt_name")
           call eval_node_compile_cvariable (en, pn, var_list)
        case ("cexpr")
           call eval_node_compile_cexpr (en, pn, var_list)
        case ("block_cexpr")
           call eval_node_compile_block_expr (en, pn, var_list, V_PDG)
        case ("conditional_cexpr")
           call eval_node_compile_conditional (en, pn, var_list, V_PDG)
        case default
           call parse_node_mismatch &
                ("grouped_cexpr|block_cexpr|conditional_cexpr|" // &
                 "pdg_code|cvariable|prt_name", pn)
        end select
        if (debug) then
```

```
         call eval_node_write (en)
         print *, "done avalue"
      end if
    end subroutine eval_node_compile_avalue
```

Compile a PDG-code expression, which is the key `PDG` with an integer expression
as argument. The procedure is analogous to `eval_node_compile_unary_function`.
⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_compile_pdg_code (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_arg
      type(eval_node_t), pointer :: en1
      type(string_t) :: key
      type(pdg_array_t) :: aval
      integer :: t
      if (debug) then
         print *, "read PDG code";  call parse_node_write (pn)
      end if
      pn_arg => parse_node_get_sub_ptr (pn, 2)
      call eval_node_compile_expr &
           (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
      t = en1%result_type
      allocate (en)
      key = "PDG"
      if (en1%type == EN_CONSTANT) then
         select case (t)
         case (V_INT)
            call pdg_i (aval, en1)
            call eval_node_init_pdg_array (en, aval)
         case default;  call eval_type_error (pn, char (key), t)
         end select
         call eval_node_final_rec (en1)
         deallocate (en1)
      else
         select case (t)
         case (V_INT);  call eval_node_set_op1_pdg (en, pdg_i)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done function"
      end if
    end subroutine eval_node_compile_pdg_code
```

This is entirely analogous to `eval_node_compile_variable`. However, PDG-
array variables occur in different contexts.

To avoid name clashes between PDG-array variables and ordinary variables,
we prepend a character (*). This is not visible to the user.
⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_node_compile_cvariable (en, pn, var_list)
```

```
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_name
      type(string_t) :: var_name
      type(var_entry_t), pointer :: var
      type(pdg_array_t), target, save :: no_aval
      logical, target, save :: unknown = .false.
      if (debug) then
         print *, "read cvariable";  call parse_node_write (pn)
      end if
      pn_name => pn
      var_name = parse_node_get_string (pn_name)
      var => var_list_get_var_ptr (var_list, var_name, V_PDG, defined=.true.)
      allocate (en)
      if (associated (var)) then
         call eval_node_init_pdg_array_ptr &
              (en, var_entry_get_name (var), var_entry_get_aval_ptr (var), &
               var_entry_get_known_ptr (var))
      else
         call parse_node_write (pn)
         call msg_error ("This PDG-array variable is undefined at this point")
         call eval_node_init_pdg_array_ptr (en, var_name, no_aval, unknown)
      end if
      if (debug) then
         call eval_node_write (en)
         print *, "done cvariable"
      end if
    end subroutine eval_node_compile_cvariable
```

String expressions

A string expression is either a string value or a concatenation of string values.

⟨*Expressions: procedures*⟩+≡
```
    recursive subroutine eval_node_compile_sexpr (en, pn, var_list)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_svalue, pn_concatenation, pn_op, pn_arg
      type(eval_node_t), pointer :: en1, en2
      type(string_t) :: string
      if (debug) then
         print *, "read sexpr";  call parse_node_write (pn)
      end if
      pn_svalue => parse_node_get_sub_ptr (pn)
      call eval_node_compile_svalue (en, pn_svalue, var_list)
      pn_concatenation => &
           parse_node_get_next_ptr (pn_svalue, tag="str_concatenation")
      do while (associated (pn_concatenation))
        pn_op => parse_node_get_sub_ptr (pn_concatenation)
        pn_arg => parse_node_get_next_ptr (pn_op)
        en1 => en
        call eval_node_compile_svalue (en2, pn_arg, var_list)
```

```
        allocate (en)
        if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
            call concat_ss (string, en1, en2)
            call eval_node_init_string (en, string)
            call eval_node_final_rec (en1)
            call eval_node_final_rec (en2)
            deallocate (en1, en2)
        else
            call eval_node_init_branch &
                 (en, var_str ("concat"), V_STR, en1, en2)
            call eval_node_set_op2_str (en, concat_ss)
        end if
        pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
    end do
    if (debug) then
        call eval_node_write (en)
        print *, "done sexpr"
    end if
end subroutine eval_node_compile_sexpr
```

A string value is a string literal, a variable, a (grouped) sexpr, a block sexpr, or
a conditional.

⟨*Expressions: procedures*⟩+≡

```
recursive subroutine eval_node_compile_svalue (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  if (debug) then
      print *, "read svalue";  call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("svariable")
      call eval_node_compile_variable (en, pn, var_list, V_STR)
  case ("sexpr")
      call eval_node_compile_sexpr (en, pn, var_list)
  case ("block_sexpr")
      call eval_node_compile_block_expr (en, pn, var_list, V_STR)
  case ("conditional_sexpr")
      call eval_node_compile_conditional (en, pn, var_list, V_STR)
  case ("sprintf_fun")
      call eval_node_compile_sprintf (en, pn, var_list)
  case ("string_literal")
      allocate (en)
      call eval_node_init_string (en, parse_node_get_string (pn))
  case default
      call parse_node_mismatch &
           ("svariable|" // &
            "grouped_sexpr|block_sexpr|conditional_sexpr|" // &
            "string_function|string_literal", pn)
  end select
  if (debug) then
      call eval_node_write (en)
      print *, "done svalue"
```

```
      end if
   end subroutine eval_node_compile_svalue
```

There is currently one string function, `sprintf`. For `sprintf`, the first argument (no brackets) is the format string, the optional arguments in brackets are the expressions or variables to be formatted.

⟨*Expressions: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_sprintf (en, pn, var_list)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     type(parse_node_t), pointer :: pn_clause, pn_key, pn_args
     type(parse_node_t), pointer :: pn_arg0
     type(eval_node_t), pointer :: en0, en1
     integer :: n_args
     type(string_t) :: key
     if (debug) then
        print *, "read sprintf_fun";  call parse_node_write (pn)
     end if
     pn_clause => parse_node_get_sub_ptr (pn)
     pn_key   => parse_node_get_sub_ptr (pn_clause)
     pn_arg0 => parse_node_get_next_ptr (pn_key)
     pn_args => parse_node_get_next_ptr (pn_clause)
     call eval_node_compile_sexpr (en0, pn_arg0, var_list)
     if (associated (pn_args)) then
        call eval_node_compile_sprintf_args (en1, pn_args, var_list, n_args)
     else
        n_args = 0
        en1 => null ()
     end if
     allocate (en)
     key = parse_node_get_key (pn_key)
     call eval_node_init_format_string (en, en0, en1, key, n_args)
     if (debug) then
        call eval_node_write (en)
        print *, "done sprintf_fun"
     end if
   end subroutine eval_node_compile_sprintf
```

⟨*Expressions: procedures*⟩+≡

```
   subroutine eval_node_compile_sprintf_args (en, pn, var_list, n_args)
     type(eval_node_t), pointer :: en
     type(parse_node_t), intent(in) :: pn
     type(var_list_t), intent(in), target :: var_list
     integer, intent(out) :: n_args
     type(parse_node_t), pointer :: pn_arg
     integer :: i
     type(eval_node_t), pointer :: en1, en2
     n_args = parse_node_get_n_sub (pn)
     en => null ()
     do i = n_args, 1, -1
        pn_arg => parse_node_get_sub_ptr (pn, i)
        select case (char (parse_node_get_rule_key (pn_arg)))
```

```
      case ("lvariable")
         call eval_node_compile_variable (en1, pn_arg, var_list, V_LOG)
      case ("svariable")
         call eval_node_compile_variable (en1, pn_arg, var_list, V_STR)
      case ("expr")
         call eval_node_compile_expr (en1, pn_arg, var_list)
      case default
         call parse_node_mismatch ("variable|svariable|lvariable|expr", pn_arg)
      end select
      if (associated (en)) then
         en2 => en
         allocate (en)
         call eval_node_init_branch &
              (en, var_str ("sprintf_arg"), V_NONE, en1, en2)
      else
         allocate (en)
         call eval_node_init_branch &
              (en, var_str ("sprintf_arg"), V_NONE, en1)
      end if
    end do
  end subroutine eval_node_compile_sprintf_args
```

Evaluation. We allocate the argument list and apply the Fortran wrapper for
the sprintf function.

⟨*Expressions: procedures*⟩+≡
```
    subroutine evaluate_sprintf (string, n_args, en_fmt, en_arg)
      type(string_t), intent(out) :: string
      integer, intent(in) :: n_args
      type(eval_node_t), pointer :: en_fmt
      type(eval_node_t), intent(in), optional, target :: en_arg
      type(eval_node_t), pointer :: en_branch, en_var
      type(sprintf_arg_t), dimension(:), allocatable :: arg
      type(string_t) :: fmt
      logical :: autoformat
      integer :: i, j, sprintf_argc
      autoformat = .not. associated (en_fmt)
      if (autoformat) fmt = ""
      if (present (en_arg)) then
         sprintf_argc = 0
         en_branch => en_arg
         do i = 1, n_args
            select case (en_branch%arg1%result_type)
               case (V_CMPLX); sprintf_argc = sprintf_argc + 2
               case default  ; sprintf_argc = sprintf_argc + 1
            end select
            en_branch => en_branch%arg2
         end do
         allocate (arg (sprintf_argc))
         j = 1
         en_branch => en_arg
         do i = 1, n_args
            en_var => en_branch%arg1
            select case (en_var%result_type)
```

577

```
              case (V_LOG)
                 call sprintf_arg_init (arg(j), en_var%lval)
                 if (autoformat) fmt = fmt // "%s "
              case (V_INT);
                 call sprintf_arg_init (arg(j), en_var%ival)
                 if (autoformat) fmt = fmt // "%i "
              case (V_REAL);
                 call sprintf_arg_init (arg(j), en_var%rval)
                 if (autoformat) fmt = fmt // "%g "
              case (V_STR)
                 call sprintf_arg_init (arg(j), en_var%sval)
                 if (autoformat) fmt = fmt // "%s "
              case (V_CMPLX)
                 call sprintf_arg_init (arg(j), real (en_var%cval, default))
                 j = j + 1
                 call sprintf_arg_init (arg(j), aimag (en_var%cval))
                 if (autoformat) fmt = fmt // "(%g + %g * I) "
              case default
                 call eval_node_write (en_var)
                 call msg_error ("sprintf is implemented " &
                      // "for logical, integer, real, and string values only")
              end select
              j = j + 1
              en_branch => en_branch%arg2
           end do
        else
           allocate (arg(0))
        end if
        if (autoformat) then
           string = sprintf (trim (fmt), arg)
        else
           string = sprintf (en_fmt%sval, arg)
        end if
     end subroutine evaluate_sprintf
```

5.7.5   Auxiliary functions for the compiler

Issue an error that the current node could not be compiled because of type mismatch:

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_type_error (pn, string, t)
      type(parse_node_t), intent(in) :: pn
      character(*), intent(in) :: string
      integer, intent(in) :: t
      type(string_t) :: type
      select case (t)
      case (V_NONE); type = "(none)"
      case (V_LOG);  type = "'logical'"
      case (V_INT);  type = "'integer'"
      case (V_REAL); type = "'real'"
      case (V_CMPLX); type = "'complex'"
      case default;  type = "(unknown)"
```

```
      end select
      call parse_node_write (pn)
      call msg_fatal (" The " // string // &
            " operation is not defined for the given argument type " // &
            char (type))
   end subroutine eval_type_error
```

If two numerics are combined, the result is integer if both arguments are integer, if one is integer and the other real or both are real, than its argument is real, otherwise complex.

⟨*Expressions: procedures*⟩+≡
```
   function numeric_result_type (t1, t2) result (t)
     integer, intent(in) :: t1, t2
     integer :: t
     if (t1 == V_INT .and. t2 == V_INT) then
        t = V_INT
      else if (t1 == V_INT .and. t2 == V_REAL) then
        t = V_REAL
      else if (t1 == V_REAL .and. t2 == V_INT) then
        t = V_REAL
      else if (t1 == V_REAL .and. t2 == V_REAL) then
        t = V_REAL
      else
        t = V_CMPLX
      end if
   end function numeric_result_type
```

5.7.6 Evaluation

Evaluation is done recursively. For leaf nodes nothing is to be done.

Evaluating particle-list functions: First, we evaluate the particle lists. If a condition is present, we assign the particle pointers of the condition node to the allocated particle entries in the parent node, keeping in mind that the observables in the variable stack used for the evaluation of the condition also contain pointers to these entries. Then, the assigned procedure is evaluated, which sets the subevent in the parent node. If required, the procedure evaluates the condition node once for each (pair of) particles to determine the result.

⟨*Expressions: procedures*⟩+≡
```
   recursive subroutine eval_node_evaluate (en)
     type(eval_node_t), intent(inout) :: en
     logical :: exist
     select case (en%type)
     case (EN_UNARY)
        if (associated (en%arg1)) then
           call eval_node_evaluate (en%arg1)
           en%value_is_known = en%arg1%value_is_known
        else
           en%value_is_known = .false.
        end if
        if (en%value_is_known) then
           select case (en%result_type)
```

579

```
            case (V_LOG);  en%lval = en% op1_log  (en%arg1)
            case (V_INT);  en%ival = en% op1_int  (en%arg1)
            case (V_REAL); en%rval = en% op1_real (en%arg1)
            case (V_CMPLX); en%cval = en% op1_cmplx (en%arg1)
            case (V_PDG);
               call en% op1_pdg  (en%aval, en%arg1)
            case (V_SEV)
               if (associated (en%arg0)) then
                  call en% op1_sev (en%pval, en%arg1, en%arg0)
               else
                  call en% op1_sev (en%pval, en%arg1)
               end if
            case (V_STR)
               call en% op1_str (en%sval, en%arg1)
            end select
      end if
   case (EN_BINARY)
      if (associated (en%arg1) .and. associated (en%arg2)) then
         call eval_node_evaluate (en%arg1)
         call eval_node_evaluate (en%arg2)
         en%value_is_known = &
               en%arg1%value_is_known .and. en%arg2%value_is_known
      else
         en%value_is_known = .false.
      end if
      if (en%value_is_known) then
         select case (en%result_type)
         case (V_LOG);  en%lval = en% op2_log  (en%arg1, en%arg2)
         case (V_INT);  en%ival = en% op2_int  (en%arg1, en%arg2)
         case (V_REAL); en%rval = en% op2_real (en%arg1, en%arg2)
         case (V_CMPLX); en%cval = en% op2_cmplx (en%arg1, en%arg2)
         case (V_PDG)
            call en% op2_pdg  (en%aval, en%arg1, en%arg2)
         case (V_SEV)
            if (associated (en%arg0)) then
               call en% op2_sev (en%pval, en%arg1, en%arg2, en%arg0)
            else
               call en% op2_sev (en%pval, en%arg1, en%arg2)
            end if
         case (V_STR)
            call en% op2_str (en%sval, en%arg1, en%arg2)
         end select
      end if
   case (EN_BLOCK)
      if (associated (en%arg1) .and. associated (en%arg0)) then
         call eval_node_evaluate (en%arg1)
         call eval_node_evaluate (en%arg0)
         en%value_is_known = en%arg0%value_is_known
      else
         en%value_is_known = .false.
      end if
      if (en%value_is_known) then
         select case (en%result_type)
         case (V_LOG);  en%lval = en%arg0%lval
```

```
         case (V_INT);  en%ival = en%arg0%ival
         case (V_REAL); en%rval = en%arg0%rval
         case (V_CMPLX); en%cval = en%arg0%cval
         case (V_PDG);  en%aval = en%arg0%aval
         case (V_SEV);  en%pval = en%arg0%pval
         case (V_STR);  en%sval = en%arg0%sval
         end select
      end if
case (EN_CONDITIONAL)
   if (associated (en%arg0)) then
      call eval_node_evaluate (en%arg0)
      en%value_is_known = en%arg0%value_is_known
   else
      en%value_is_known = .false.
   end if
   if (en%arg0%value_is_known) then
      if (en%arg0%lval) then
         call eval_node_evaluate (en%arg1)
         en%value_is_known = en%arg1%value_is_known
         if (en%value_is_known) then
            select case (en%result_type)
            case (V_LOG);  en%lval = en%arg1%lval
            case (V_INT);  en%ival = en%arg1%ival
            case (V_REAL); en%rval = en%arg1%rval
            case (V_CMPLX); en%cval = en%arg1%cval
            case (V_PDG);  en%aval = en%arg1%aval
            case (V_SEV);  en%pval = en%arg1%pval
            case (V_STR);  en%sval = en%arg1%sval
            end select
         end if
      else
         call eval_node_evaluate (en%arg2)
         en%value_is_known = en%arg2%value_is_known
         if (en%value_is_known) then
            select case (en%result_type)
            case (V_LOG);  en%lval = en%arg2%lval
            case (V_INT);  en%ival = en%arg2%ival
            case (V_REAL); en%rval = en%arg2%rval
            case (V_CMPLX); en%cval = en%arg2%cval
            case (V_PDG);  en%aval = en%arg2%aval
            case (V_SEV);  en%pval = en%arg2%pval
            case (V_STR);  en%sval = en%arg2%sval
            end select
         end if
      end if
   end if
   ! call eval_node_write_rec (en)    !!! Debugging
case (EN_RECORD_CMD)
   exist = .true.
   en%lval = .false.
   call eval_node_evaluate (en%arg0)
   if (en%arg0%value_is_known) then
      if (associated (en%arg1)) then
         call eval_node_evaluate (en%arg1)
```

```fortran
if (en%arg1%value_is_known) then
    if (associated (en%arg2)) then
        call eval_node_evaluate (en%arg2)
        if (en%arg2%value_is_known) then
            if (associated (en%arg3)) then
                call eval_node_evaluate (en%arg3)
                if (en%arg3%value_is_known) then
                    if (associated (en%arg4)) then
                        call eval_node_evaluate (en%arg4)
                        if (en%arg4%value_is_known) then
                            if (associated (en%rval)) then
                                call analysis_record_data (en%arg0%sval, &
                                    en%arg1%rval, en%arg2%rval, &
                                    en%arg3%rval, en%arg4%rval, &
                                    weight=en%rval, exist=exist, &
                                    success=en%lval)
                            else
                                call analysis_record_data (en%arg0%sval, &
                                    en%arg1%rval, en%arg2%rval, &
                                    en%arg3%rval, en%arg4%rval, &
                                    exist=exist, success=en%lval)
                            end if
                        end if
                    else
                        if (associated (en%rval)) then
                            call analysis_record_data (en%arg0%sval, &
                                en%arg1%rval, en%arg2%rval, &
                                en%arg3%rval, &
                                weight=en%rval, exist=exist, &
                                success=en%lval)
                        else
                            call analysis_record_data (en%arg0%sval, &
                                en%arg1%rval, en%arg2%rval, &
                                en%arg3%rval, &
                                exist=exist, success=en%lval)
                        end if
                    end if
                end if
            else
                if (associated (en%rval)) then
                    call analysis_record_data (en%arg0%sval, &
                        en%arg1%rval, en%arg2%rval, &
                        weight=en%rval, exist=exist, &
                        success=en%lval)
                else
                    call analysis_record_data (en%arg0%sval, &
                        en%arg1%rval, en%arg2%rval, &
                        exist=exist, success=en%lval)
                end if
            end if
        end if
    else
        if (associated (en%rval)) then
            call analysis_record_data (en%arg0%sval, &
```

```
                         en%arg1%rval, &
                         weight=en%rval, exist=exist, success=en%lval)
                  else
                     call analysis_record_data (en%arg0%sval, &
                         en%arg1%rval, &
                         exist=exist, success=en%lval)
                  end if
               end if
            end if
      else
         if (associated (en%rval)) then
            call analysis_record_data (en%arg0%sval, 1._default, &
                weight=en%rval, exist=exist, success=en%lval)
         else
            call analysis_record_data (en%arg0%sval, 1._default, &
                exist=exist, success=en%lval)
         end if
      end if
      if (.not. exist) then
         call msg_error ("Analysis object '" // char (en%arg0%sval) &
             // "' is undefined")
         en%arg0%value_is_known = .false.
      end if
   end if
case (EN_OBS1_INT)
   en%ival = en% obs1_int (en%prt1)
   en%value_is_known = .true.
case (EN_OBS2_INT)
   en%ival = en% obs2_int (en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_OBS1_REAL)
   en%rval = en% obs1_real (en%prt1)
   en%value_is_known = .true.
case (EN_OBS2_REAL)
   en%rval = en% obs2_real (en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_UOBS1_INT)
   en%ival = user_obs_int_p (en%arg0, en%prt1)
   en%value_is_known = .true.
case (EN_UOBS2_INT)
   en%ival = user_obs_int_pp (en%arg0, en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_UOBS1_REAL)
   en%rval = user_obs_real_p (en%arg0, en%prt1)
   en%value_is_known = .true.
case (EN_UOBS2_REAL)
   en%rval = user_obs_real_pp (en%arg0, en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_PRT_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = en%arg1%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
```

```
                 en%arg0%prt1 => en%prt1
                 call en% op1_sev (en%pval, en%arg1, en%arg0)
             else
                 call en% op1_sev (en%pval, en%arg1)
             end if
         end if
case (EN_PRT_FUN_BINARY)
    call eval_node_evaluate (en%arg1)
    call eval_node_evaluate (en%arg2)
    en%value_is_known = &
         en%arg1%value_is_known .and. en%arg2%value_is_known
    if (en%value_is_known) then
         if (associated (en%arg0)) then
             en%arg0%index => en%index
             en%arg0%prt1 => en%prt1
             en%arg0%prt2 => en%prt2
             call en% op2_sev (en%pval, en%arg1, en%arg2, en%arg0)
         else
             call en% op2_sev (en%pval, en%arg1, en%arg2)
         end if
    end if
case (EN_EVAL_FUN_UNARY)
    call eval_node_evaluate (en%arg1)
    en%value_is_known = subevt_is_nonempty (en%arg1%pval)
    if (en%value_is_known) then
         en%arg0%index => en%index
         en%index = 1
         en%arg0%prt1 => en%prt1
         en%prt1 = subevt_get_prt (en%arg1%pval, 1)
         call eval_node_evaluate (en%arg0)
         en%rval = en%arg0%rval
    end if
case (EN_EVAL_FUN_BINARY)
    call eval_node_evaluate (en%arg1)
    call eval_node_evaluate (en%arg2)
    en%value_is_known = &
         subevt_is_nonempty (en%arg1%pval) .and. &
         subevt_is_nonempty (en%arg2%pval)
    if (en%value_is_known) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         en%arg0%prt2 => en%prt2
         en%index = 1
         call eval_pp (en%arg1, en%arg2, en%arg0, en%rval, en%value_is_known)
    end if
case (EN_LOG_FUN_UNARY)
    call eval_node_evaluate (en%arg1)
    en%value_is_known = .true.
    if (en%value_is_known) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         en%lval = en% op1_cut (en%arg1, en%arg0)
    end if
case (EN_LOG_FUN_BINARY)
```

```
      call eval_node_evaluate (en%arg1)
      call eval_node_evaluate (en%arg2)
      en%value_is_known = .true.
      if (en%value_is_known) then
          en%arg0%index => en%index
          en%arg0%prt1 => en%prt1
          en%arg0%prt2 => en%prt2
          en%lval = en% op2_cut (en%arg1, en%arg2, en%arg0)
      end if
case (EN_INT_FUN_UNARY)
      call eval_node_evaluate (en%arg1)
      en%value_is_known = en%arg1%value_is_known
      if (en%value_is_known) then
          if (associated (en%arg0)) then
              en%arg0%index => en%index
              en%arg0%prt1 => en%prt1
              call en% op1_evi (en%ival, en%arg1, en%arg0)
          else
              call en% op1_evi (en%ival, en%arg1)
          end if
      end if
case (EN_INT_FUN_BINARY)
      call eval_node_evaluate (en%arg1)
      call eval_node_evaluate (en%arg2)
      en%value_is_known = &
          en%arg1%value_is_known .and. &
          en%arg2%value_is_known
      if (en%value_is_known) then
          if (associated (en%arg0)) then
              en%arg0%index => en%index
              en%arg0%prt1 => en%prt1
              en%arg0%prt2 => en%prt2
              call en% op2_evi (en%ival, en%arg1, en%arg2, en%arg0)
          else
              call en% op2_evi (en%ival, en%arg1, en%arg2)
          end if
      end if
case (EN_REAL_FUN_UNARY)
      call eval_node_evaluate (en%arg1)
      en%value_is_known = en%arg1%value_is_known
      if (en%value_is_known) then
          if (associated (en%arg0)) then
              en%arg0%index => en%index
              en%arg0%prt1 => en%prt1
              call en% op1_evr (en%rval, en%arg1, en%arg0)
          else
              call en% op1_evr (en%rval, en%arg1)
          end if
      end if
case (EN_REAL_FUN_BINARY)
      call eval_node_evaluate (en%arg1)
      call eval_node_evaluate (en%arg2)
      en%value_is_known = &
          en%arg1%value_is_known .and. &
```

```
                 en%arg2%value_is_known
        if (en%value_is_known) then
            if (associated (en%arg0)) then
                en%arg0%index => en%index
                en%arg0%prt1 => en%prt1
                en%arg0%prt2 => en%prt2
                call en% op2_evr (en%rval, en%arg1, en%arg2, en%arg0)
            else
                call en% op2_evr (en%rval, en%arg1, en%arg2)
            end if
        end if
    case (EN_FORMAT_STR)
        if (associated (en%arg0)) then
            call eval_node_evaluate (en%arg0)
            en%value_is_known = en%arg0%value_is_known
        else
            en%value_is_known = .true.
        end if
        if (associated (en%arg1)) then
            call eval_node_evaluate (en%arg1)
            en%value_is_known = &
                en%value_is_known .and. en%arg1%value_is_known
            if (en%value_is_known) then
                call evaluate_sprintf (en%sval, en%ival, en%arg0, en%arg1)
            end if
        else
            if (en%value_is_known) then
                call evaluate_sprintf (en%sval, en%ival, en%arg0)
            end if
        end if
    end select
    if (debug) then
        print *, "evaluated"
        call eval_node_write (en)
    end if
end subroutine eval_node_evaluate
```

5.7.7   Evaluation syntax

We have two different flavors of the syntax: with and without particles.

⟨*Expressions: variables*⟩+≡
```
    type(syntax_t), target, save :: syntax_expr
    type(syntax_t), target, save :: syntax_pexpr
```

These are for testing only and may be removed:

⟨*Expressions: public*⟩≡
```
    public :: syntax_expr_init
    public :: syntax_pexpr_init
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine syntax_expr_init ()
      type(ifile_t) :: ifile
```

```
      call define_expr_syntax (ifile, particles=.false., analysis=.false.)
      call syntax_init (syntax_expr, ifile)
      call ifile_final (ifile)
    end subroutine syntax_expr_init

    subroutine syntax_pexpr_init ()
      type(ifile_t) :: ifile
      call define_expr_syntax (ifile, particles=.true., analysis=.false.)
      call syntax_init (syntax_pexpr, ifile)
      call ifile_final (ifile)
    end subroutine syntax_pexpr_init
```

⟨*Expressions: public*⟩+≡
```
    public :: syntax_expr_final
    public :: syntax_pexpr_final
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine syntax_expr_final ()
      call syntax_final (syntax_expr)
    end subroutine syntax_expr_final

    subroutine syntax_pexpr_final ()
      call syntax_final (syntax_pexpr)
    end subroutine syntax_pexpr_final
```

⟨*Expressions: public*⟩+≡
```
    public :: syntax_pexpr_write
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine syntax_pexpr_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_pexpr, unit)
    end subroutine syntax_pexpr_write
```

⟨*Expressions: public*⟩+≡
```
    public :: define_expr_syntax
```

Numeric expressions.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_expr_syntax (ifile, particles, analysis)
      type(ifile_t), intent(inout) :: ifile
      logical, intent(in) :: particles, analysis
      type(string_t) :: numeric_pexpr
      type(string_t) :: var_plist, var_alias
      if (particles) then
         numeric_pexpr = " | numeric_pexpr"
         var_plist = " | var_plist"
         var_alias = " | var_alias"
      else
         numeric_pexpr = ""
         var_plist = ""
         var_alias = ""
      end if
      call ifile_append (ifile, "SEQ expr = subexpr addition*")
```

```
call ifile_append (ifile, "ALT subexpr = addition | term")
call ifile_append (ifile, "SEQ addition = plus_or_minus term")
call ifile_append (ifile, "SEQ term = factor multiplication*")
call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
call ifile_append (ifile, "SEQ factor = value exponentiation?")
call ifile_append (ifile, "SEQ exponentiation = to_the value")
call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
call ifile_append (ifile, "ALT to_the = '^' | '**'")
call ifile_append (ifile, "KEY '+'")
call ifile_append (ifile, "KEY '-'")
call ifile_append (ifile, "KEY '*'")
call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "KEY '^'")
call ifile_append (ifile, "KEY '**'")
call ifile_append (ifile, "ALT value = signed_value | unsigned_value")
call ifile_append (ifile, "SEQ signed_value = '-' unsigned_value")
call ifile_append (ifile, "ALT unsigned_value = " // &
     "numeric_value | constant | variable | " // &
     "result | user_observable | " // &
     "grouped_expr | block_expr | conditional_expr | " // &
     "unary_function | binary_function" // &
     numeric_pexpr)
call ifile_append (ifile, "ALT numeric_value = integer_value | " &
     // "real_value | complex_value")
call ifile_append (ifile, "SEQ integer_value = integer_literal unit_expr?")
call ifile_append (ifile, "SEQ real_value = real_literal unit_expr?")
call ifile_append (ifile, "SEQ complex_value = complex_literal unit_expr?")
call ifile_append (ifile, "INT integer_literal")
call ifile_append (ifile, "REA real_literal")
call ifile_append (ifile, "COM complex_literal")
call ifile_append (ifile, "SEQ unit_expr = unit unit_power?")
call ifile_append (ifile, "ALT unit = " // &
     "TeV | GeV | MeV | keV | eV | meV | " // &
     "nbarn | pbarn | fbarn | abarn | " // &
     "rad | mrad | degree | '%'")
call ifile_append (ifile, "KEY TeV")
call ifile_append (ifile, "KEY GeV")
call ifile_append (ifile, "KEY MeV")
call ifile_append (ifile, "KEY keV")
call ifile_append (ifile, "KEY eV")
call ifile_append (ifile, "KEY meV")
call ifile_append (ifile, "KEY nbarn")
call ifile_append (ifile, "KEY pbarn")
call ifile_append (ifile, "KEY fbarn")
call ifile_append (ifile, "KEY abarn")
call ifile_append (ifile, "KEY rad")
call ifile_append (ifile, "KEY mrad")
call ifile_append (ifile, "KEY degree")
call ifile_append (ifile, "KEY '%'")
call ifile_append (ifile, "SEQ unit_power = '^' frac_expr")
call ifile_append (ifile, "ALT frac_expr = frac | grouped_frac")
call ifile_append (ifile, "GRO grouped_frac = ( frac_expr )")
call ifile_append (ifile, "SEQ frac = signed_int div?")
```

```
call ifile_append (ifile, "ALT signed_int = " &
     // "neg_int | pos_int | integer_literal")
call ifile_append (ifile, "SEQ neg_int = '-' integer_literal")
call ifile_append (ifile, "SEQ pos_int = '+' integer_literal")
call ifile_append (ifile, "SEQ div = '/' integer_literal")
call ifile_append (ifile, "ALT constant = pi | I")
call ifile_append (ifile, "KEY pi")
call ifile_append (ifile, "KEY I")
call ifile_append (ifile, "IDE variable")
call ifile_append (ifile, "SEQ result = result_key result_arg")
call ifile_append (ifile, "ALT result_key = " // &
     "num_id | integral | error")
call ifile_append (ifile, "SEQ user_observable = user_obs user_arg")
call ifile_append (ifile, "KEY user_obs")
call ifile_append (ifile, "ARG user_arg = ( sexpr )")
call ifile_append (ifile, "KEY num_id")
call ifile_append (ifile, "KEY integral")
call ifile_append (ifile, "KEY error")
call ifile_append (ifile, "GRO result_arg = ( process_id )")
call ifile_append (ifile, "IDE process_id")
call ifile_append (ifile, "SEQ unary_function = fun_unary function_arg1")
call ifile_append (ifile, "SEQ binary_function = fun_binary function_arg2")
call ifile_append (ifile, "ALT fun_unary = " // &
     "complex | real | int | nint | floor | ceiling | abs | sgn | " // &
     "sqrt | exp | log | log10 | " // &
     "sin | cos | tan | asin | acos | atan | " // &
     "sinh | cosh | tanh")
call ifile_append (ifile, "KEY complex")
call ifile_append (ifile, "KEY real")
call ifile_append (ifile, "KEY int")
call ifile_append (ifile, "KEY nint")
call ifile_append (ifile, "KEY floor")
call ifile_append (ifile, "KEY ceiling")
call ifile_append (ifile, "KEY abs")
call ifile_append (ifile, "KEY sgn")
call ifile_append (ifile, "KEY sqrt")
call ifile_append (ifile, "KEY exp")
call ifile_append (ifile, "KEY log")
call ifile_append (ifile, "KEY log10")
call ifile_append (ifile, "KEY sin")
call ifile_append (ifile, "KEY cos")
call ifile_append (ifile, "KEY tan")
call ifile_append (ifile, "KEY asin")
call ifile_append (ifile, "KEY acos")
call ifile_append (ifile, "KEY atan")
call ifile_append (ifile, "KEY sinh")
call ifile_append (ifile, "KEY cosh")
call ifile_append (ifile, "KEY tanh")
call ifile_append (ifile, "ALT fun_binary = max | min | mod | modulo")
call ifile_append (ifile, "KEY max")
call ifile_append (ifile, "KEY min")
call ifile_append (ifile, "KEY mod")
call ifile_append (ifile, "KEY modulo")
call ifile_append (ifile, "ARG function_arg1 = ( expr )")
```

```
    call ifile_append (ifile, "ARG function_arg2 = ( expr, expr )")
    call ifile_append (ifile, "GRO grouped_expr = ( expr )")
    call ifile_append (ifile, "SEQ block_expr = let var_spec in expr")
    call ifile_append (ifile, "KEY let")
    call ifile_append (ifile, "ALT var_spec = " // &
         "var_num | var_int | var_real | var_complex | " // &
         "var_logical" // var_plist // var_alias // " | var_string")
    call ifile_append (ifile, "SEQ var_num = var_name '=' expr")
    call ifile_append (ifile, "SEQ var_int = int var_name '=' expr")
    call ifile_append (ifile, "SEQ var_real = real var_name '=' expr")
    call ifile_append (ifile, "SEQ var_complex = complex var_name '=' complex_expr")
    call ifile_append (ifile, "ALT complex_expr = " // &
         "cexpr_real | cexpr_complex")
    call ifile_append (ifile, "ARG cexpr_complex = ( expr, expr )")
    call ifile_append (ifile, "SEQ cexpr_real = expr")
    call ifile_append (ifile, "IDE var_name")
    call ifile_append (ifile, "KEY '='")
    call ifile_append (ifile, "KEY in")
    call ifile_append (ifile, "SEQ conditional_expr = " // &
         "if lexpr then expr maybe_elsif_expr maybe_else_expr endif")
    call ifile_append (ifile, "SEQ maybe_elsif_expr = elsif_expr*")
    call ifile_append (ifile, "SEQ maybe_else_expr = else_expr?")
    call ifile_append (ifile, "SEQ elsif_expr = elsif lexpr then expr")
    call ifile_append (ifile, "SEQ else_expr = else expr")
    call ifile_append (ifile, "KEY if")
    call ifile_append (ifile, "KEY then")
    call ifile_append (ifile, "KEY elsif")
    call ifile_append (ifile, "KEY else")
    call ifile_append (ifile, "KEY endif")
    call define_lexpr_syntax (ifile, particles, analysis)
    call define_sexpr_syntax (ifile)
    if (particles) then
       call define_pexpr_syntax (ifile)
       call define_cexpr_syntax (ifile)
       call define_var_plist_syntax (ifile)
       call define_var_alias_syntax (ifile)
       call define_numeric_pexpr_syntax (ifile)
       call define_logical_pexpr_syntax (ifile)
    end if

  end subroutine define_expr_syntax
```

Logical expressions.
⟨*Expressions: procedures*⟩+≡
```
  subroutine define_lexpr_syntax (ifile, particles, analysis)
    type(ifile_t), intent(inout) :: ifile
    logical, intent(in) :: particles, analysis
    type(string_t) :: logical_pexpr, record_cmd
    if (particles) then
       logical_pexpr = " | logical_pexpr"
    else
       logical_pexpr = ""
    end if
    if (analysis) then
```

```
      record_cmd = " | record_cmd"
else
      record_cmd = ""
end if
call ifile_append (ifile, "SEQ lexpr = lsinglet lsequel*")
call ifile_append (ifile, "SEQ lsequel = ';' lsinglet")
call ifile_append (ifile, "SEQ lsinglet = lterm alternative*")
call ifile_append (ifile, "SEQ alternative = or lterm")
call ifile_append (ifile, "SEQ lterm = lvalue coincidence*")
call ifile_append (ifile, "SEQ coincidence = and lvalue")
call ifile_append (ifile, "KEY ';'")
call ifile_append (ifile, "KEY or")
call ifile_append (ifile, "KEY and")
call ifile_append (ifile, "ALT lvalue = " // &
      "true | false | lvariable | negation | " // &
      "grouped_lexpr | block_lexpr | conditional_lexpr | " // &
      "compared_expr | compared_sexpr" // &
      logical_pexpr //  record_cmd)
call ifile_append (ifile, "KEY true")
call ifile_append (ifile, "KEY false")
call ifile_append (ifile, "SEQ lvariable = '?' alt_lvariable")
call ifile_append (ifile, "KEY '?'")
call ifile_append (ifile, "ALT alt_lvariable = variable | grouped_lexpr")
call ifile_append (ifile, "SEQ negation = not lvalue")
call ifile_append (ifile, "KEY not")
call ifile_append (ifile, "GRO grouped_lexpr = ( lexpr )")
call ifile_append (ifile, "SEQ block_lexpr = let var_spec in lexpr")
call ifile_append (ifile, "ALT var_logical = " // &
      "var_logical_new | var_logical_spec")
call ifile_append (ifile, "SEQ var_logical_new = logical var_logical_spec")
call ifile_append (ifile, "KEY logical")
call ifile_append (ifile, "SEQ var_logical_spec = '?' var_name = lexpr")
call ifile_append (ifile, "SEQ conditional_lexpr = " // &
      "if lexpr then lexpr maybe_elsif_lexpr maybe_else_lexpr endif")
call ifile_append (ifile, "SEQ maybe_elsif_lexpr = elsif_lexpr*")
call ifile_append (ifile, "SEQ maybe_else_lexpr = else_lexpr?")
call ifile_append (ifile, "SEQ elsif_lexpr = elsif lexpr then lexpr")
call ifile_append (ifile, "SEQ else_lexpr = else lexpr")
call ifile_append (ifile, "SEQ compared_expr = expr comparison+")
call ifile_append (ifile, "SEQ comparison = compare expr")
call ifile_append (ifile, "ALT compare = " // &
      "'<' | '>' | '<=' | '>=' | '==' | '<>'")
call ifile_append (ifile, "KEY '<'")
call ifile_append (ifile, "KEY '>'")
call ifile_append (ifile, "KEY '<='")
call ifile_append (ifile, "KEY '>='")
call ifile_append (ifile, "KEY '=='")
call ifile_append (ifile, "KEY '<>'")
call ifile_append (ifile, "SEQ compared_sexpr = sexpr str_comparison+")
call ifile_append (ifile, "SEQ str_comparison = str_compare sexpr")
call ifile_append (ifile, "ALT str_compare = '==' | '<>'")
if (analysis) then
      call ifile_append (ifile, "SEQ record_cmd = " // &
            "record_key analysis_tag record_arg?")
```

```
        call ifile_append (ifile, "ALT record_key = " // &
              "record | record_unweighted | record_excess")
        call ifile_append (ifile, "KEY record")
        call ifile_append (ifile, "KEY record_unweighted")
        call ifile_append (ifile, "KEY record_excess")
        call ifile_append (ifile, "ALT analysis_tag = analysis_id | sexpr")
        call ifile_append (ifile, "IDE analysis_id")
        call ifile_append (ifile, "ARG record_arg = ( expr+ )")
      end if
    end subroutine define_lexpr_syntax
```

String expressions.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_sexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ sexpr = svalue str_concatenation*")
      call ifile_append (ifile, "SEQ str_concatenation = '&' svalue")
      call ifile_append (ifile, "KEY '&'")
      call ifile_append (ifile, "ALT svalue = " // &
            "grouped_sexpr | block_sexpr | conditional_sexpr | " // &
            "svariable | string_function | string_literal")
      call ifile_append (ifile, "GRO grouped_sexpr = ( sexpr )")
      call ifile_append (ifile, "SEQ block_sexpr = let var_spec in sexpr")
      call ifile_append (ifile, "SEQ conditional_sexpr = " // &
            "if lexpr then sexpr maybe_elsif_sexpr maybe_else_sexpr endif")
      call ifile_append (ifile, "SEQ maybe_elsif_sexpr = elsif_sexpr*")
      call ifile_append (ifile, "SEQ maybe_else_sexpr = else_sexpr?")
      call ifile_append (ifile, "SEQ elsif_sexpr = elsif lexpr then sexpr")
      call ifile_append (ifile, "SEQ else_sexpr = else sexpr")
      call ifile_append (ifile, "SEQ svariable = '$' alt_svariable")
      call ifile_append (ifile, "KEY '$'")
      call ifile_append (ifile, "ALT alt_svariable = variable | grouped_sexpr")
      call ifile_append (ifile, "ALT var_string = " // &
            "var_string_new | var_string_spec")
      call ifile_append (ifile, "SEQ var_string_new = string var_string_spec")
      call ifile_append (ifile, "KEY string")
      call ifile_append (ifile, "SEQ var_string_spec = '$' var_name = sexpr") ! $
      call ifile_append (ifile, "ALT string_function = sprintf_fun")
      call ifile_append (ifile, "SEQ sprintf_fun = sprintf_clause sprintf_args?")
      call ifile_append (ifile, "SEQ sprintf_clause = sprintf sexpr")
      call ifile_append (ifile, "KEY sprintf")
      call ifile_append (ifile, "ARG sprintf_args = ( sprintf_arg* )")
      call ifile_append (ifile, "ALT sprintf_arg = " &
            // "lvariable | svariable | expr")
      call ifile_append (ifile, "QUO string_literal = '""'...'""'")
    end subroutine define_sexpr_syntax
```

Expressions that evaluate to subevents.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_pexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ pexpr = pterm pconcatenation*")
      call ifile_append (ifile, "SEQ pconcatenation = '&' pterm")
```

```
! call ifile_append (ifile, "KEY '&'")    !!! (Key exists already)
call ifile_append (ifile, "SEQ pterm = pvalue pcombination*")
call ifile_append (ifile, "SEQ pcombination = '+' pvalue")
! call ifile_append (ifile, "KEY '+'")    !!! (Key exists already)
call ifile_append (ifile, "ALT pvalue = " // &
     "pexpr_src | pvariable | " // &
     "grouped_pexpr | block_pexpr | conditional_pexpr | " // &
     "prt_function")
call ifile_append (ifile, "SEQ pexpr_src = prefix_cexpr")
call ifile_append (ifile, "ALT prefix_cexpr = " // &
     "incoming_prt | outgoing_prt")
call ifile_append (ifile, "SEQ incoming_prt = incoming cexpr")
call ifile_append (ifile, "KEY incoming")
call ifile_append (ifile, "SEQ outgoing_prt = cexpr")
call ifile_append (ifile, "SEQ pvariable = '@' alt_pvariable")
call ifile_append (ifile, "KEY '@'")
call ifile_append (ifile, "ALT alt_pvariable = variable | grouped_pexpr")
call ifile_append (ifile, "GRO grouped_pexpr = '[' pexpr ']'")
call ifile_append (ifile, "SEQ block_pexpr = let var_spec in pexpr")
call ifile_append (ifile, "SEQ conditional_pexpr = " // &
     "if lexpr then pexpr maybe_elsif_pexpr maybe_else_pexpr endif")
call ifile_append (ifile, "SEQ maybe_elsif_pexpr = elsif_pexpr*")
call ifile_append (ifile, "SEQ maybe_else_pexpr = else_pexpr?")
call ifile_append (ifile, "SEQ elsif_pexpr = elsif lexpr then pexpr")
call ifile_append (ifile, "SEQ else_pexpr = else pexpr")
call ifile_append (ifile, "ALT prt_function = " // &
     "join_fun | combine_fun | collect_fun | cluster_fun | " // &
     "select_fun | extract_fun | sort_fun")
call ifile_append (ifile, "SEQ join_fun = join_clause pargs2")
call ifile_append (ifile, "SEQ combine_fun = combine_clause pargs2")
call ifile_append (ifile, "SEQ collect_fun = collect_clause pargs1")
call ifile_append (ifile, "SEQ cluster_fun = cluster_clause pargs1")
call ifile_append (ifile, "SEQ select_fun = select_clause pargs1")
call ifile_append (ifile, "SEQ extract_fun = extract_clause pargs1")
call ifile_append (ifile, "SEQ sort_fun = sort_clause pargs1")
call ifile_append (ifile, "SEQ join_clause = join condition?")
call ifile_append (ifile, "SEQ combine_clause = combine condition?")
call ifile_append (ifile, "SEQ collect_clause = collect condition?")
call ifile_append (ifile, "SEQ cluster_clause = cluster condition?")
call ifile_append (ifile, "SEQ select_clause = select condition?")
call ifile_append (ifile, "SEQ extract_clause = extract position?")
call ifile_append (ifile, "SEQ sort_clause = sort criterion?")
call ifile_append (ifile, "KEY join")
call ifile_append (ifile, "KEY combine")
call ifile_append (ifile, "KEY collect")
call ifile_append (ifile, "KEY cluster")
call ifile_append (ifile, "KEY select")
call ifile_append (ifile, "SEQ condition = if lexpr")
call ifile_append (ifile, "KEY extract")
call ifile_append (ifile, "SEQ position = index expr")
call ifile_append (ifile, "KEY sort")
call ifile_append (ifile, "SEQ criterion = by expr")
call ifile_append (ifile, "KEY index")
call ifile_append (ifile, "KEY by")
```

```
      call ifile_append (ifile, "ARG pargs2 = '[' pexpr, pexpr ']'")
      call ifile_append (ifile, "ARG pargs1 = '[' pexpr, pexpr? ']'")
    end subroutine define_pexpr_syntax
```

Expressions that evaluate to PDG-code arrays.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_cexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ cexpr = avalue concatenation*")
      call ifile_append (ifile, "SEQ concatenation = ':' avalue")
      call ifile_append (ifile, "KEY ':'")
      call ifile_append (ifile, "ALT avalue = " // &
           "grouped_cexpr | block_cexpr | conditional_cexpr | " // &
           "variable | pdg_code | prt_name")
      call ifile_append (ifile, "GRO grouped_cexpr = ( cexpr )")
      call ifile_append (ifile, "SEQ block_cexpr = let var_spec in cexpr")
      call ifile_append (ifile, "SEQ conditional_cexpr = " // &
           "if lexpr then cexpr maybe_elsif_cexpr maybe_else_cexpr endif")
      call ifile_append (ifile, "SEQ maybe_elsif_cexpr = elsif_cexpr*")
      call ifile_append (ifile, "SEQ maybe_else_cexpr = else_cexpr?")
      call ifile_append (ifile, "SEQ elsif_cexpr = elsif lexpr then cexpr")
      call ifile_append (ifile, "SEQ else_cexpr = else cexpr")
      call ifile_append (ifile, "SEQ pdg_code = pdg pdg_arg")
      call ifile_append (ifile, "KEY pdg")
      call ifile_append (ifile, "ARG pdg_arg = ( expr )")
      call ifile_append (ifile, "QUO prt_name = '""'...'""'")
    end subroutine define_cexpr_syntax
```

Extra variable types.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_var_plist_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "ALT var_plist = var_plist_new | var_plist_spec")
      call ifile_append (ifile, "SEQ var_plist_new = subevt var_plist_spec")
      call ifile_append (ifile, "KEY subevt")
      call ifile_append (ifile, "SEQ var_plist_spec = '@' var_name '=' pexpr")
    end subroutine define_var_plist_syntax

    subroutine define_var_alias_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ var_alias = alias var_name '=' cexpr")
      call ifile_append (ifile, "KEY alias")
    end subroutine define_var_alias_syntax
```

Particle-list expressions that evaluate to numeric values

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_numeric_pexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "ALT numeric_pexpr = " &
           // "eval_fun | count_fun | event_shape_fun")
      call ifile_append (ifile, "SEQ eval_fun = eval expr pargs1")
      call ifile_append (ifile, "SEQ count_fun = count_clause pargs1")
```

```
      call ifile_append (ifile, "SEQ count_clause = count condition?")
      call ifile_append (ifile, "KEY eval")
      call ifile_append (ifile, "KEY count")
      call ifile_append (ifile, "ALT event_shape_fun = user_event_fun")
      call ifile_append (ifile, "SEQ user_event_fun = " &
           // "user_event_shape user_arg pargs1")
      call ifile_append (ifile, "KEY user_event_shape")
    end subroutine define_numeric_pexpr_syntax
```

Particle-list functions that evaluate to logical values.

⟨*Expressions: procedures*⟩+≡
```
    subroutine define_logical_pexpr_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "ALT logical_pexpr = " // &
           "all_fun | any_fun | no_fun | user_cut_fun")
      call ifile_append (ifile, "SEQ all_fun = all lexpr pargs1")
      call ifile_append (ifile, "SEQ any_fun = any lexpr pargs1")
      call ifile_append (ifile, "SEQ no_fun = no lexpr pargs1")
      call ifile_append (ifile, "KEY all")
      call ifile_append (ifile, "KEY any")
      call ifile_append (ifile, "KEY no")
      call ifile_append (ifile, "SEQ user_cut_fun = user_cut user_arg pargs1")
      call ifile_append (ifile, "KEY user_cut")
    end subroutine define_logical_pexpr_syntax
```

All characters that can occur in expressions (apart from alphanumeric).

⟨*Expressions: procedures*⟩+≡
```
    subroutine lexer_init_eval_tree (lexer, particles)
      type(lexer_t), intent(out) :: lexer
      logical, intent(in) :: particles
      type(keyword_list_t), pointer :: keyword_list
      if (particles) then
         keyword_list => syntax_get_keyword_list_ptr (syntax_pexpr)
      else
         keyword_list => syntax_get_keyword_list_ptr (syntax_expr)
      end if
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "()[],;:&%?$@", &
           special_class = (/ "+-*/^", "<>=~ " /) , &
           keyword_list = keyword_list)
    end subroutine lexer_init_eval_tree
```

5.7.8 Set up appropriate parse trees

Parse an input stream as a specific flavor of expression. The appropriate expression syntax has to be available.

⟨*Expressions: public*⟩+≡
```
    public :: parse_tree_init_expr
```

```
      public :: parse_tree_init_lexpr
      public :: parse_tree_init_pexpr
      public :: parse_tree_init_cexpr
      public :: parse_tree_init_sexpr
```
⟨*Expressions: procedures*⟩+≡
```
    subroutine parse_tree_init_expr (parse_tree, stream, particles)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      logical, intent(in) :: particles
      type(lexer_t) :: lexer
      call lexer_init_eval_tree (lexer, particles)
      call lexer_assign_stream (lexer, stream)
      if (particles) then
         call parse_tree_init &
              (parse_tree, syntax_pexpr, lexer, var_str ("expr"))
      else
         call parse_tree_init &
              (parse_tree, syntax_expr, lexer, var_str ("expr"))
      end if
      call lexer_final (lexer)
    end subroutine parse_tree_init_expr

    subroutine parse_tree_init_lexpr (parse_tree, stream, particles)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      logical, intent(in) :: particles
      type(lexer_t) :: lexer
      call lexer_init_eval_tree (lexer, particles)
      call lexer_assign_stream (lexer, stream)
      if (particles) then
         call parse_tree_init &
              (parse_tree, syntax_pexpr, lexer, var_str ("lexpr"))
      else
         call parse_tree_init &
              (parse_tree, syntax_expr, lexer, var_str ("lexpr"))
      end if
      call lexer_final (lexer)
    end subroutine parse_tree_init_lexpr

    subroutine parse_tree_init_pexpr (parse_tree, stream)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      type(lexer_t) :: lexer
      call lexer_init_eval_tree (lexer, .true.)
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init &
           (parse_tree, syntax_pexpr, lexer, var_str ("pexpr"))
      call lexer_final (lexer)
    end subroutine parse_tree_init_pexpr

    subroutine parse_tree_init_cexpr (parse_tree, stream)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      type(lexer_t) :: lexer
```

```
      call lexer_init_eval_tree (lexer, .true.)
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init &
           (parse_tree, syntax_pexpr, lexer, var_str ("cexpr"))
      call lexer_final (lexer)
    end subroutine parse_tree_init_cexpr

    subroutine parse_tree_init_sexpr (parse_tree, stream, particles)
      type(parse_tree_t), intent(out) :: parse_tree
      type(stream_t), intent(inout), target :: stream
      logical, intent(in) :: particles
      type(lexer_t) :: lexer
      call lexer_init_eval_tree (lexer, particles)
      call lexer_assign_stream (lexer, stream)
      if (particles) then
         call parse_tree_init &
              (parse_tree, syntax_pexpr, lexer, var_str ("sexpr"))
      else
         call parse_tree_init &
              (parse_tree, syntax_expr, lexer, var_str ("sexpr"))
      end if
      call lexer_final (lexer)
    end subroutine parse_tree_init_sexpr
```

5.7.9 The evaluation tree

The evaluation tree contains the initial variable list and the root node.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_t
```

⟨*Expressions: types*⟩+≡
```
    type :: eval_tree_t
      private
      type(var_list_t) :: var_list
      type(eval_node_t), pointer :: root => null ()
     contains
     ⟨Expressions: eval tree: TBP⟩
    end type eval_tree_t
```

Init from stream, using a temporary parse tree.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_init_stream &
         (eval_tree, stream, var_list, subevt, result_type)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(stream_t), intent(inout), target :: stream
      type(var_list_t), intent(in), target :: var_list
      type(subevt_t), intent(in), target, optional :: subevt
      integer, intent(in), optional :: result_type
      type(parse_tree_t) :: parse_tree
      type(parse_node_t), pointer :: nd_root
      integer :: type
      type = V_REAL;  if (present (result_type))  type = result_type
```

```
      select case (type)
      case (V_INT, V_REAL, V_CMPLX)
         call parse_tree_init_expr (parse_tree, stream, present (subevt))
      case (V_LOG)
         call parse_tree_init_lexpr (parse_tree, stream, present (subevt))
      case (V_SEV)
         call parse_tree_init_pexpr (parse_tree, stream)
      case (V_PDG)
         call parse_tree_init_cexpr (parse_tree, stream)
      case (V_STR)
         call parse_tree_init_sexpr (parse_tree, stream, present (subevt))
      end select
      nd_root => parse_tree_get_root_ptr (parse_tree)
      if (associated (nd_root)) then
         select case (type)
         case (V_INT, V_REAL, V_CMPLX)
            call eval_tree_init_expr (eval_tree, nd_root, var_list, subevt)
         case (V_LOG)
            call eval_tree_init_lexpr (eval_tree, nd_root, var_list, subevt)
         case (V_SEV)
            call eval_tree_init_pexpr (eval_tree, nd_root, var_list, subevt)
         case (V_PDG)
            call eval_tree_init_cexpr (eval_tree, nd_root, var_list, subevt)
         case (V_STR)
            call eval_tree_init_sexpr (eval_tree, nd_root, var_list, subevt)
         end select
      end if
      call parse_tree_final (parse_tree)
    end subroutine eval_tree_init_stream
```

API: Init from a given parse-tree node. If we evaluate an expression that contains particle-list references, the original subevent has to be supplied. The initial variable list is optional.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_init_expr
    public :: eval_tree_init_lexpr
    public :: eval_tree_init_pexpr
    public :: eval_tree_init_cexpr
    public :: eval_tree_init_sexpr
```
⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_init_expr &
        (eval_tree, parse_node, var_list, subevt)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(subevt_t), intent(in), optional, target :: subevt
      call eval_tree_set_var_list (eval_tree, var_list, subevt)
      call eval_node_compile_expr &
          (eval_tree%root, parse_node, eval_tree%var_list)
    end subroutine eval_tree_init_expr

    subroutine eval_tree_init_lexpr &
        (eval_tree, parse_node, var_list, subevt)
```

```
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    call eval_tree_set_var_list (eval_tree, var_list, subevt)
    call eval_node_compile_lexpr &
         (eval_tree%root, parse_node, eval_tree%var_list)
  end subroutine eval_tree_init_lexpr

  subroutine eval_tree_init_pexpr &
       (eval_tree, parse_node, var_list, subevt)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    call eval_tree_set_var_list (eval_tree, var_list, subevt)
    call eval_node_compile_pexpr &
         (eval_tree%root, parse_node, eval_tree%var_list)
  end subroutine eval_tree_init_pexpr

  subroutine eval_tree_init_cexpr &
       (eval_tree, parse_node, var_list, subevt)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    call eval_tree_set_var_list (eval_tree, var_list, subevt)
    call eval_node_compile_cexpr &
         (eval_tree%root, parse_node, eval_tree%var_list)
  end subroutine eval_tree_init_cexpr

  subroutine eval_tree_init_sexpr &
       (eval_tree, parse_node, var_list, subevt)
    type(eval_tree_t), intent(out), target :: eval_tree
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    call eval_tree_set_var_list (eval_tree, var_list, subevt)
    call eval_node_compile_sexpr &
         (eval_tree%root, parse_node, eval_tree%var_list)
  end subroutine eval_tree_init_sexpr
```

This extra API function handles numerical constant expressions only. The only nontrivial part is the optional unit.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_init_numeric_value
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_init_numeric_value (eval_tree, parse_node)
      type(eval_tree_t), intent(out), target :: eval_tree
      type(parse_node_t), intent(in), target :: parse_node
      call eval_node_compile_numeric_value (eval_tree%root, parse_node)
    end subroutine eval_tree_init_numeric_value
```

599

Initialize the variable list with the initial one; if a particle list is provided, add a pointer to this as variable @evt. If the event weight is provided as a real-valued target, add a pointer to it as well.

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_set_var_list &
        (eval_tree, var_list, subevt)
      type(eval_tree_t), intent(inout), target :: eval_tree
      type(var_list_t), intent(in), target :: var_list
      type(subevt_t), intent(in), optional, target :: subevt
      logical, save, target :: known = .true.
      call var_list_link (eval_tree%var_list, var_list)
      if (present (subevt))  call var_list_append_subevt_ptr &
          (eval_tree%var_list, var_str ("@evt"), subevt, known, &
          intrinsic=.true.)
    end subroutine eval_tree_set_var_list
```

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_final
```
⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_final (eval_tree)
      type(eval_tree_t), intent(inout) :: eval_tree
      call var_list_final (eval_tree%var_list)
      if (associated (eval_tree%root)) then
          call eval_node_final_rec (eval_tree%root)
          deallocate (eval_tree%root)
      end if
    end subroutine eval_tree_final
```

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_evaluate
```
⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_evaluate (eval_tree)
      type(eval_tree_t), intent(inout) :: eval_tree
      if (associated (eval_tree%root)) then
          call eval_node_evaluate (eval_tree%root)
      end if
    end subroutine eval_tree_evaluate
```

Check if the eval tree is allocated.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_is_defined
```
⟨*Expressions: procedures*⟩+≡
```
    function eval_tree_is_defined (eval_tree) result (flag)
      logical :: flag
      type(eval_tree_t), intent(in) :: eval_tree
      flag = associated (eval_tree%root)
    end function eval_tree_is_defined
```

Check if the eval tree result is constant.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_is_constant
```

600

⟨Expressions: procedures⟩+≡

```
function eval_tree_is_constant (eval_tree) result (flag)
  logical :: flag
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
      flag = eval_tree%root%type == EN_CONSTANT
  else
      flag = .false.
  end if
end function eval_tree_is_constant
```

Insert a conversion node at the root, if necessary (only for real/int conversion)

⟨Expressions: public⟩+≡

```
public :: eval_tree_convert_result
```

⟨Expressions: procedures⟩+≡

```
subroutine eval_tree_convert_result (eval_tree, result_type)
  type(eval_tree_t), intent(inout) :: eval_tree
  integer, intent(in) :: result_type
  if (associated (eval_tree%root)) then
      call insert_conversion_node (eval_tree%root, result_type)
  end if
end subroutine eval_tree_convert_result
```

Return the value of the top node, after evaluation. If the tree is empty, return the type of `V_NONE`. When extracting the value, no check for existence is done. For numeric values, the functions are safe against real/integer mismatch.

⟨Expressions: public⟩+≡

```
public :: eval_tree_get_result_type
public :: eval_tree_result_is_known
public :: eval_tree_result_is_known_ptr
public :: eval_tree_get_log
public :: eval_tree_get_int
public :: eval_tree_get_real
public :: eval_tree_get_cmplx
public :: eval_tree_get_pdg_array
public :: eval_tree_get_subevt
public :: eval_tree_get_string
```

⟨Expressions: procedures⟩+≡

```
function eval_tree_get_result_type (eval_tree) result (type)
  integer :: type
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
      type = eval_tree%root%result_type
  else
      type = V_NONE
  end if
end function eval_tree_get_result_type

function eval_tree_result_is_known (eval_tree) result (flag)
  logical :: flag
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
```

```fortran
      select case (eval_tree%root%result_type)
      case (V_LOG, V_INT, V_REAL)
         flag = eval_tree%root%value_is_known
      case default
         flag = .true.
      end select
  else
     flag = .false.
  end if
end function eval_tree_result_is_known

function eval_tree_result_is_known_ptr (eval_tree) result (ptr)
  logical, pointer :: ptr
  type(eval_tree_t), intent(in) :: eval_tree
  logical, target, save :: known = .true.
  if (associated (eval_tree%root)) then
     select case (eval_tree%root%result_type)
     case (V_LOG, V_INT, V_REAL)
        ptr => eval_tree%root%value_is_known
     case default
        ptr => known
     end select
  else
     ptr => null ()
  end if
end function eval_tree_result_is_known_ptr

function eval_tree_get_log (eval_tree) result (lval)
  logical :: lval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root))  lval = eval_tree%root%lval
end function eval_tree_get_log

function eval_tree_get_int (eval_tree) result (ival)
  integer :: ival
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     select case (eval_tree%root%result_type)
     case (V_INT);   ival = eval_tree%root%ival
     case (V_REAL);  ival = eval_tree%root%rval
     case (V_CMPLX); ival = eval_tree%root%cval
     end select
  end if
end function eval_tree_get_int

function eval_tree_get_real (eval_tree) result (rval)
  real(default) :: rval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     select case (eval_tree%root%result_type)
     case (V_REAL);  rval = eval_tree%root%rval
     case (V_INT);   rval = eval_tree%root%ival
     case (V_CMPLX);  rval = eval_tree%root%cval
     end select
```

```
      end if
    end function eval_tree_get_real

    function eval_tree_get_cmplx (eval_tree) result (cval)
      complex(default) :: cval
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         select case (eval_tree%root%result_type)
         case (V_CMPLX); cval = eval_tree%root%cval
         case (V_REAL); cval = eval_tree%root%rval
         case (V_INT);  cval = eval_tree%root%ival
         end select
      end if
    end function eval_tree_get_cmplx

    function eval_tree_get_pdg_array (eval_tree) result (aval)
      type(pdg_array_t) :: aval
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         aval = eval_tree%root%aval
      end if
    end function eval_tree_get_pdg_array

    function eval_tree_get_subevt (eval_tree) result (pval)
      type(subevt_t) :: pval
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         pval = eval_tree%root%pval
      end if
    end function eval_tree_get_subevt

    function eval_tree_get_string (eval_tree) result (sval)
      type(string_t) :: sval
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
         sval = eval_tree%root%sval
      end if
    end function eval_tree_get_string
```

Return a pointer to the value of the top node.

⟨*Expressions: public*⟩+≡
```
    public :: eval_tree_get_log_ptr
    public :: eval_tree_get_int_ptr
    public :: eval_tree_get_real_ptr
    public :: eval_tree_get_cmplx_ptr
    public :: eval_tree_get_subevt_ptr
    public :: eval_tree_get_pdg_array_ptr
    public :: eval_tree_get_string_ptr
```

⟨*Expressions: procedures*⟩+≡
```
    function eval_tree_get_log_ptr (eval_tree) result (lval)
      logical, pointer :: lval
      type(eval_tree_t), intent(in) :: eval_tree
      if (associated (eval_tree%root)) then
```

```
      lval => eval_tree%root%lval
   else
      lval => null ()
   end if
end function eval_tree_get_log_ptr

function eval_tree_get_int_ptr (eval_tree) result (ival)
  integer, pointer :: ival
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     ival => eval_tree%root%ival
  else
     ival => null ()
  end if
end function eval_tree_get_int_ptr

function eval_tree_get_real_ptr (eval_tree) result (rval)
  real(default), pointer :: rval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     rval => eval_tree%root%rval
  else
     rval => null ()
  end if
end function eval_tree_get_real_ptr

function eval_tree_get_cmplx_ptr (eval_tree) result (cval)
  complex(default), pointer :: cval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     cval => eval_tree%root%cval
  else
     cval => null ()
  end if
end function eval_tree_get_cmplx_ptr

function eval_tree_get_subevt_ptr (eval_tree) result (pval)
  type(subevt_t), pointer :: pval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     pval => eval_tree%root%pval
  else
     pval => null ()
  end if
end function eval_tree_get_subevt_ptr

function eval_tree_get_pdg_array_ptr (eval_tree) result (aval)
  type(pdg_array_t), pointer :: aval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     aval => eval_tree%root%aval
  else
     aval => null ()
  end if
```

```
      end function eval_tree_get_pdg_array_ptr

      function eval_tree_get_string_ptr (eval_tree) result (sval)
        type(string_t), pointer :: sval
        type(eval_tree_t), intent(in) :: eval_tree
        if (associated (eval_tree%root)) then
           sval => eval_tree%root%sval
        else
           sval => null ()
        end if
      end function eval_tree_get_string_ptr
```

⟨*Expressions: public*⟩+≡
```
      public :: eval_tree_write
```

⟨*Expressions: eval tree: TBP*⟩≡
```
      procedure :: write => eval_tree_write
```

⟨*Expressions: procedures*⟩+≡
```
      subroutine eval_tree_write (eval_tree, unit, write_var_list)
        class(eval_tree_t), intent(in) :: eval_tree
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: write_var_list
        integer :: u
        logical :: vl
        u = output_unit (unit);  if (u < 0)  return
        vl = .false.;  if (present (write_var_list))  vl = write_var_list
        write (u, "(1x,A)") "Evaluation tree:"
        if (associated (eval_tree%root)) then
           call eval_node_write_rec (eval_tree%root, unit)
        else
           write (u, "(3x,A)") "[empty]"
        end if
        if (vl)  call var_list_write (eval_tree%var_list, unit)
      end subroutine eval_tree_write
```

Use the written representation for generating an MD5 sum:

⟨*Expressions: public*⟩+≡
```
      public :: eval_tree_get_md5sum
```

⟨*Expressions: procedures*⟩+≡
```
      function eval_tree_get_md5sum (eval_tree) result (md5sum_et)
        character(32) :: md5sum_et
        type(eval_tree_t), intent(in) :: eval_tree
        integer :: u
        u = free_unit ()
        open (unit = u, status = "scratch", action = "readwrite")
        call eval_tree_write (eval_tree, unit=u)
        rewind (u)
        md5sum_et = md5sum (u)
        close (u)
      end function eval_tree_get_md5sum
```

5.7.10 Direct evaluation

These procedures create an eval tree and evaluate it on-the-fly, returning only
the final value. The evaluation must yield a well-defined value, unless the
`is_known` flag is present, which will be set accordingly.

⟨*Expressions: public*⟩+≡
```
    public :: eval_log
    public :: eval_int
    public :: eval_real
    public :: eval_cmplx
    public :: eval_subevt
    public :: eval_pdg_array
    public :: eval_string
```
⟨*Expressions: procedures*⟩+≡
```
    function eval_log &
         (parse_node, var_list, subevt, is_known) result (lval)
      logical :: lval
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(subevt_t), intent(in), optional, target :: subevt
      logical, intent(out), optional :: is_known
      type(eval_tree_t), target :: eval_tree
      call eval_tree_init_lexpr &
           (eval_tree, parse_node, var_list, subevt)
      call eval_tree_evaluate (eval_tree)
      if (eval_tree_result_is_known (eval_tree)) then
         if (present (is_known))  is_known = .true.
         lval = eval_tree_get_log (eval_tree)
      else if (present (is_known)) then
         is_known = .false.
      else
         call eval_tree_unknown (eval_tree, parse_node)
         lval = .false.
      end if
      call eval_tree_final (eval_tree)
    end function eval_log

    function eval_int &
         (parse_node, var_list, subevt, is_known) result (ival)
      integer :: ival
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(subevt_t), intent(in), optional, target :: subevt
      logical, intent(out), optional :: is_known
      type(eval_tree_t), target :: eval_tree
      call eval_tree_init_expr &
           (eval_tree, parse_node, var_list, subevt)
      call eval_tree_evaluate (eval_tree)
      if (eval_tree_result_is_known (eval_tree)) then
         if (present (is_known))  is_known = .true.
         ival = eval_tree_get_int (eval_tree)
      else if (present (is_known)) then
         is_known = .false.
      else
```

```
      call eval_tree_unknown (eval_tree, parse_node)
      ival = 0
   end if
   call eval_tree_final (eval_tree)
end function eval_int

function eval_real &
     (parse_node, var_list, subevt, is_known) result (rval)
  real(default) :: rval
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_expr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     rval = eval_tree_get_real (eval_tree)
  else if (present (is_known)) then
     is_known = .false.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     rval = 0
  end if
  call eval_tree_final (eval_tree)
end function eval_real

function eval_cmplx &
     (parse_node, var_list, subevt, is_known) result (cval)
  complex(default) :: cval
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_expr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     cval = eval_tree_get_cmplx (eval_tree)
  else if (present (is_known)) then
     is_known = .false.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     cval = 0
  end if
  call eval_tree_final (eval_tree)
end function eval_cmplx

function eval_subevt &
     (parse_node, var_list, subevt, is_known) result (pval)
```

```
    type(subevt_t) :: pval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_pexpr &
         (eval_tree, parse_node, var_list, subevt)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
       if (present (is_known))  is_known = .true.
       pval = eval_tree_get_subevt (eval_tree)
    else if (present (is_known)) then
       is_known = .false.
    else
       call eval_tree_unknown (eval_tree, parse_node)
    end if
    call eval_tree_final (eval_tree)
  end function eval_subevt

  function eval_pdg_array &
       (parse_node, var_list, subevt, is_known) result (aval)
    type(pdg_array_t) :: aval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_cexpr &
         (eval_tree, parse_node, var_list, subevt)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
       if (present (is_known))  is_known = .true.
       aval = eval_tree_get_pdg_array (eval_tree)
    else if (present (is_known)) then
       is_known = .false.
    else
       call eval_tree_unknown (eval_tree, parse_node)
    end if
    call eval_tree_final (eval_tree)
  end function eval_pdg_array

  function eval_string &
       (parse_node, var_list, subevt, is_known) result (sval)
    type(string_t) :: sval
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    logical, intent(out), optional :: is_known
    type(eval_tree_t), target :: eval_tree
    call eval_tree_init_sexpr &
         (eval_tree, parse_node, var_list, subevt)
    call eval_tree_evaluate (eval_tree)
    if (eval_tree_result_is_known (eval_tree)) then
```

```
         if (present (is_known))  is_known = .true.
         sval = eval_tree_get_string (eval_tree)
      else if (present (is_known)) then
         is_known = .false.
      else
         call eval_tree_unknown (eval_tree, parse_node)
         sval = ""
      end if
      call eval_tree_final (eval_tree)
    end function eval_string
```

Here is a variant that returns numeric values of all possible kinds, the appropriate kind to be selected later:

⟨*Expressions: public*⟩+≡
```
    public :: eval_numeric
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_numeric &
         (parse_node, var_list, subevt, ival, rval, cval, &
          is_known, result_type)
      type(parse_node_t), intent(in), target :: parse_node
      type(var_list_t), intent(in), target :: var_list
      type(subevt_t), intent(in), optional, target :: subevt
      integer, intent(out), optional :: ival
      real(default), intent(out), optional :: rval
      complex(default), intent(out), optional :: cval
      logical, intent(out), optional :: is_known
      integer, intent(out), optional :: result_type
      type(eval_tree_t), target :: eval_tree
      call eval_tree_init_expr &
           (eval_tree, parse_node, var_list, subevt)
      call eval_tree_evaluate (eval_tree)
      if (eval_tree_result_is_known (eval_tree)) then
         if (present (ival))  ival = eval_tree_get_int (eval_tree)
         if (present (rval))  rval = eval_tree_get_real (eval_tree)
         if (present (cval))  cval = eval_tree_get_cmplx (eval_tree)
         if (present (is_known))  is_known = .true.
      else
         call eval_tree_unknown (eval_tree, parse_node)
         if (present (ival))  ival = 0
         if (present (rval))  rval = 0
         if (present (cval))  cval = 0
         if (present (is_known))  is_known = .false.
      end if
      if (present (result_type))  &
           result_type = eval_tree_get_result_type (eval_tree)
      call eval_tree_final (eval_tree)
    end subroutine eval_numeric
```

Error message with debugging info:

⟨*Expressions: procedures*⟩+≡
```
    subroutine eval_tree_unknown (eval_tree, parse_node)
      type(eval_tree_t), intent(in) :: eval_tree
```

```
      type(parse_node_t), intent(in) :: parse_node
      call parse_node_write_rec (parse_node)
      call eval_tree_write (eval_tree)
      call msg_error ("Evaluation yields an undefined result, inserting default")
    end subroutine eval_tree_unknown
```

5.7.11   Unit tests

⟨*Expressions: public*⟩+≡
```
    public :: expressions_test
```

⟨*Expressions: procedures*⟩+≡
```
    subroutine expressions_test (u, results)
      integer, intent(in) :: u
      type (test_results_t), intent(inout) :: results
    ⟨Expressions: execute tests⟩
    end subroutine expressions_test
```

Testing the routines of the expressions module. First a simple unary observable
and the node evaluation.

⟨*Expressions: execute tests*⟩≡
```
    call test (expressions_1, "expressions_1", &
        "check simple observable", &
        u, results)
```

⟨*Expressions: tests*⟩≡
```
    subroutine expressions_1 (u)
      integer, intent(in) :: u
      type(var_list_t), pointer :: var_list => null ()
      type(eval_node_t), pointer :: node => null ()
      type(prt_t), pointer :: prt => null ()
      type(var_entry_t), pointer :: var => null ()

      write (u, "(A)")  "* Test output: Expressions"
      write (u, "(A)")  "*   Purpose: test simple observable and node evaluation"
      write (u, "(A)")

      write (u, "(A)")  "* Setting a unary observable:"
      write (u, "(A)")

      allocate (var_list)
      allocate (prt)
      call var_list_set_observables_unary (var_list, prt)
      call var_list_write (var_list, u)
      var => var_list_get_var_ptr (var_list, var_str ("PDG"))

      write (u, "(A)")  "* Evaluating the observable node:"
      write (u, "(A)")

      allocate (node)
      call eval_node_init_obs (node, var)
      call eval_node_write (node, u)
```

610

```
    write (u, "(A)")  "* Cleanup"
    write (u, "(A)")

    call eval_node_final_rec (node)
    deallocate (node)
    call var_list_final (var_list)
    deallocate (var_list)
    deallocate (prt)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: expressions_1"

  end subroutine expressions_1
```

Parse a complicated expression, transfer it to a parse tree and evaluate.

⟨*Expressions: execute tests*⟩+≡
```
    call test (expressions_2, "expressions_2", &
        "check expression transfer to parse tree", &
        u, results)
```

⟨*Expressions: tests*⟩+≡
```
  subroutine expressions_2 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(eval_tree_t) :: eval_tree
    type(string_t) :: expr_text
    type(var_list_t), pointer :: var_list => null ()

    write (u, "(A)")  "* Test output: Expressions"
    write (u, "(A)")  "*   Purpose: test parse routines"
    write (u, "(A)")

    call syntax_expr_init ()
    call syntax_write (syntax_expr, u)
    allocate (var_list)
    call var_list_append_real (var_list, var_str ("tolerance"), 0._default)
    call var_list_append_real (var_list, var_str ("x"), -5._default)
    call var_list_append_int  (var_list, var_str ("foo"), -27)
    call var_list_append_real (var_list, var_str ("mb"), 4._default)
    expr_text = &
        "let real twopi = 2 * pi in" // &
        "  twopi * sqrt (25.d0 - mb^2)" // &
        "  / (let int mb_or_0 = max (mb, 0) in" // &
        "        1 + (if -1 TeV <= x < mb_or_0 then abs(x) else x endif))"
    call ifile_append (ifile, expr_text)
    call stream_init (stream, ifile)
    call var_list_write (var_list, unit = u)
    call eval_tree_init_stream (eval_tree, stream, var_list=var_list)
    call eval_tree_evaluate (eval_tree)
    call eval_tree_write (eval_tree, u)

    write (u, "(A)")  "* Input string:"
    write (u, "(A,A)")  "      ", char (expr_text)
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call stream_final (stream)
      call ifile_final (ifile)
      call eval_tree_final (eval_tree)
      call var_list_final (var_list)
      deallocate (var_list)
      call syntax_expr_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: expressions_2"

    end subroutine expressions_2
```

Test a subevent expression.

⟨*Expressions: execute tests*⟩+≡
```
      call test (expressions_3, "expressions_3", &
           "check subevent expressions", &
           u, results)
```
⟨*Expressions: tests*⟩+≡
```
    subroutine expressions_3 (u)
      integer, intent(in) :: u
      type(subevt_t) :: subevt

      write (u, "(A)")   "* Test output: Expressions"
      write (u, "(A)")   "*   Purpose: test subevent expressions"
      write (u, "(A)")

      write (u, "(A)")   "* Initialize subevent:"
      write (u, "(A)")

      call subevt_init (subevt)
      call subevt_reset (subevt, 1)
      call subevt_set_incoming (subevt, 1, &
           22, vector4_moving (1.e3_default, 1.e3_default, 1), &
           0._default, [2])
      call subevt_write (subevt, u)
      call subevt_reset (subevt, 4)
      call subevt_reset (subevt, 3)
      call subevt_set_incoming (subevt, 1, &
           21, vector4_moving (1.e3_default, 1.e3_default, 3), &
           0._default, [1])
      call subevt_polarize (subevt, 1, -1)
      call subevt_set_outgoing (subevt, 2, &
           1, vector4_moving (0._default, 1.e3_default, 3), &
           -1.e6_default, [7])
      call subevt_set_composite (subevt, 3, &
           vector4_moving (-1.e3_default, 0._default, 3), &
           (/ 2, 7 /))
      call subevt_write (subevt, u)

      write (u, "(A)")
```

612

```
          write (u, "(A)")  "* Test output end: expressions_3"

      end subroutine expressions_3
```

Test expressions from a PDG array.

⟨Expressions: execute tests⟩+≡
```
      call test (expressions_4, "expressions_4", &
           "check pdg array expressions", &
           u, results)
```

⟨Expressions: tests⟩+≡
```
      subroutine expressions_4 (u)
        integer, intent(in) :: u
        type(subevt_t), target :: subevt
        type(string_t) :: expr_text
        type(ifile_t) :: ifile
        type(stream_t) :: stream
        type(eval_tree_t) :: eval_tree
        type(var_list_t), pointer :: var_list => null ()
        type(pdg_array_t) :: aval

        write (u, "(A)")  "* Test output: Expressions"
        write (u, "(A)")  "*   Purpose: test pdg array expressions"
        write (u, "(A)")

        write (u, "(A)")  "* Initialization:"
        write (u, "(A)")

        call syntax_pexpr_init ()
        call syntax_write (syntax_pexpr, u)
        allocate (var_list)
        call var_list_append_real (var_list, var_str ("tolerance"), 0._default)
        aval = 0
        call var_list_append_pdg_array (var_list, var_str ("particle"), aval)
        aval = [11,-11]
        call var_list_append_pdg_array (var_list, var_str ("lepton"), aval)
        aval = 22
        call var_list_append_pdg_array (var_list, var_str ("photon"), aval)
        aval = 1
        call var_list_append_pdg_array (var_list, var_str ("u"), aval)
        call subevt_init (subevt)
        call subevt_reset (subevt, 6)
        call subevt_set_incoming (subevt, 1, &
             1, vector4_moving (1._default, 1._default, 1), 0._default)
        call subevt_set_incoming (subevt, 2, &
             -1, vector4_moving (2._default, 2._default, 1), 0._default)
        call subevt_set_outgoing (subevt, 3, &
             22, vector4_moving (3._default, 3._default, 1), 0._default)
        call subevt_set_outgoing (subevt, 4, &
             22, vector4_moving (4._default, 4._default, 1), 0._default)
        call subevt_set_outgoing (subevt, 5, &
             11, vector4_moving (5._default, 5._default, 1), 0._default)
        call subevt_set_outgoing (subevt, 6, &
             -11, vector4_moving (6._default, 6._default, 1), 0._default)
```

```
      write (u, "(A)")
      write (u, "(A)")  "* Expression:"
      expr_text = &
           "let alias quark = pdg(1):pdg(2):pdg(3) in" // &
           "  any E > 3 GeV " // &
           "    [sort by - Pt " // &
           "        [select if Index < 6 " // &
           "            [photon:pdg(-11):pdg(3):quark " // &
           "             & incoming particle]]]" // &
           "  and" // &
           "  eval Theta [extract index -1 [photon]] > 45 degree" // &
           "  and" // &
           "  count [incoming photon] * 3 > 0"
      write (u, "(A,A)")  "     ", char (expr_text)
      write (u, "(A)")

      write (u, "(A)")
      write (u, "(A)")  "* Extract the evaluation tree:"
      write (u, "(A)")

      call ifile_append (ifile, expr_text)
      call stream_init (stream, ifile)
      call eval_tree_init_stream (eval_tree, stream, var_list, subevt, V_LOG)
      call eval_tree_write (eval_tree, u)
      call eval_tree_evaluate (eval_tree)

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate the tree:"
      write (u, "(A)")

      call eval_tree_write (eval_tree, u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"
      write (u, "(A)")

      call stream_final (stream)
      call ifile_final (ifile)
      call eval_tree_final (eval_tree)
      call var_list_final (var_list)
      deallocate (var_list)
      call syntax_pexpr_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: expressions_4"

    end subroutine expressions_4
```

Chapter 6

Physics Models

A model object represents a physics model. It contains a table of particle data, a list of parameters, and a vertex table. The list of parameters is a variable list which includes the real parameters (which are pointers to the particle data table) and PDG array variables for the particles themselves. The vertex list is used for phase-space generation, not for calculating the matrix element.

Model definitions are read from model files. Therefore, this module contains a parser for model files. The parameter definitions (derived parameters) are Sindarin expressions.

The models, as read from file, are stored in a model library which is a simple list of model definitions. For setting up a process object we should make a copy (an instance) of a model, which gets the current parameter values from the global variable list.

6.1 Model module

⟨models.f90⟩≡
  ⟨*File header*⟩

  module models

    use iso_c_binding !NODEP!
  ⟨*Use kinds*⟩
    use kinds, only: i8, i32 !NODEP!
    use kinds, only: c_default_float !NODEP!
  ⟨*Use strings*⟩
    use limits, only: FMT_19, VERTEX_TABLE_SCALE_FACTOR !NODEP!
  ⟨*Use file utils*⟩
    use unit_tests
    use md5
    use os_interface
    use hashes, only: hash
    use diagnostics !NODEP!
    use ifiles
    use syntax_rules
    use lexers
    use parser

```
    use pdg_arrays
    use variables
    use expressions
```

⟨*Standard module head*⟩

⟨*Models: public*⟩

⟨*Models: parameters*⟩

⟨*Models: types*⟩

⟨*Models: interfaces*⟩

⟨*Models: variables*⟩

```
contains
```

⟨*Models: procedures*⟩

⟨*Models: tests*⟩

```
end module models
```

6.1.1 Physics Parameters

A parameter has a name, a value. Derived parameters also have a definition in
terms of other parameters, which is stored as an `eval_tree`. External parame-
ters are set by an external program.

The `pn` component is a pointer to the parameter definition inside the model
parse tree. It allows us to recreate the `eval_tree` when making copies (in-
stances) of the parameter object.

⟨*Models: parameters*⟩≡
```
    integer, parameter :: PAR_NONE = 0
    integer, parameter :: PAR_INDEPENDENT = 1, PAR_DERIVED = 2
    integer, parameter :: PAR_EXTERNAL = 3
```

⟨*Models: public*⟩≡
```
    public :: parameter_t
```
⟨*Models: types*⟩≡
```
    type :: parameter_t
       private
       integer :: type  = PAR_NONE
       type(string_t) :: name
       real(default) :: value = 0
       type(parse_node_t), pointer :: pn => null ()
       type(eval_tree_t) :: eval_tree
    end type parameter_t
```

Initialization depends on parameter type. Independent parameters are initial-
ized by a constant value or a constant numerical expression (which may contain

616

a unit). Derived parameters are initialized by an arbitrary numerical expression, which makes use of the current variable list. The expression is evaluated by the function `parameter_reset`.

⟨*Models: procedures*⟩≡
```
    subroutine parameter_init_independent_value (par, name, value)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      real(default), intent(in) :: value
      par%type = PAR_INDEPENDENT
      par%name = name
      par%value = value
    end subroutine parameter_init_independent_value

    subroutine parameter_init_independent (par, name, pn)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      type(parse_node_t), intent(in), target :: pn
      par%type = PAR_INDEPENDENT
      par%name = name
      par%pn => pn
      call eval_tree_init_numeric_value (par%eval_tree, pn)
      par%value = eval_tree_get_real (par%eval_tree)
    end subroutine parameter_init_independent

    subroutine parameter_init_derived (par, name, pn, var_list)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      par%type = PAR_DERIVED
      par%name = name
      par%pn => pn
      call eval_tree_init_expr (par%eval_tree, pn, var_list=var_list)
      call parameter_reset_derived (par)
    end subroutine parameter_init_derived

    subroutine parameter_init_external (par, name)
      type(parameter_t), intent(out) :: par
      type(string_t), intent(in) :: name
      par%type = PAR_EXTERNAL
      par%name = name
    end subroutine parameter_init_external
```

The finalizer is needed for the evaluation tree in the definition.

⟨*Models: procedures*⟩+≡
```
    subroutine parameter_final (par)
      type(parameter_t), intent(inout) :: par
      call eval_tree_final (par%eval_tree)
    end subroutine parameter_final
```

All derived parameters should be recalculated if some independent parameters have changed:

⟨*Models: procedures*⟩+≡

```
    subroutine parameter_reset_derived (par)
      type(parameter_t), intent(inout) :: par
      select case (par%type)
      case (PAR_DERIVED)
         call eval_tree_evaluate (par%eval_tree)
         par%value = eval_tree_get_real (par%eval_tree)
      end select
    end subroutine parameter_reset_derived
```

Direct access to the parameter value:

⟨*Models: procedures*⟩+≡
```
    function parameter_get_value_ptr (par) result (val)
      real(default), pointer :: val
      type(parameter_t), intent(in), target :: par
      val => par%value
    end function parameter_get_value_ptr
```

Output. [We should have a formula format for the eval tree, suitable for input and output!]

⟨*Models: procedures*⟩+≡
```
    subroutine parameter_write (par, unit, write_defs)
      type(parameter_t), intent(in) :: par
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: write_defs
      logical :: defs
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      defs = .false.;  if (present (write_defs))  defs = write_defs
      select case (par%type)
      case (PAR_INDEPENDENT)
         write (u, "(3x,A)", advance="no")  "parameter"
      case (PAR_DERIVED)
         write (u, "(3x,A)", advance="no")  "derived"
      case (PAR_EXTERNAL)
         write (u, "(3x,A)", advance="no")  "external"
      end select
      write (u, "(1x,A,1x,A)", advance="no")  char (par%name), "= "
      write (u, "(" // FMT_19 // ")", advance="no") par%value
      select case (par%type)
      case (PAR_DERIVED)
         if (defs) then
            call eval_tree_write (par%eval_tree, unit)
         else
            write (u, *)
         end if
      case default
         write (u, *)
      end select
    end subroutine parameter_write
```

Screen output variant.

⟨*Models: procedures*⟩+≡
```

```
```

```
subroutine parameter_show_independent (par, l, u)
  type(parameter_t), intent(in) :: par
  integer, intent(in) :: l, u
  character(len=l) :: buffer
  select case (par%type)
  case (PAR_INDEPENDENT)
     buffer = par%name
     write (u, "(4x,A,1x,'=',1x," // FMT_19 // ")")  buffer, par%value
  end select
end subroutine parameter_show_independent

subroutine parameter_show_derived (par, l, u)
  type(parameter_t), intent(in) :: par
  integer, intent(in) :: l, u
  character(len=l) :: buffer
  select case (par%type)
  case (PAR_DERIVED)
     buffer = par%name
     write (u, "(4x,A,1x,'=',1x," // FMT_19 // ")")  buffer, par%value
  end select
end subroutine parameter_show_derived

subroutine parameter_show_external (par, l, u)
  type(parameter_t), intent(in) :: par
  integer, intent(in) :: l, u
  character(len=l) :: buffer
  select case (par%type)
  case (PAR_EXTERNAL)
     buffer = par%name
     write (u, "(4x,A,1x,'=',1x," // FMT_19 // ")")  buffer, par%value
  end select
end subroutine parameter_show_external
```

6.1.2   Particle codes

Let us define a few particle codes independent of the model.

SM fermions:

⟨*Models: types*⟩+≡
```
integer, parameter, public :: ELECTRON = 11
```

Gauge bosons:

⟨*Models: types*⟩+≡
```
integer, parameter, public :: GLUON = 21
integer, parameter, public :: PHOTON = 22
integer, parameter, public :: Z_BOSON = 23
integer, parameter, public :: W_BOSON = 24
```

Hadrons:

⟨*Models: types*⟩+≡
```
integer, parameter, public :: PROTON = 2212
```

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: PION = 111
    integer, parameter, public :: PIPLUS = 211
    integer, parameter, public :: PIMINUS = - PIPLUS
```

Hadron remnants (internal)

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: HADRON_REMNANT = 90
    integer, parameter, public :: HADRON_REMNANT_SINGLET = 91
    integer, parameter, public :: HADRON_REMNANT_TRIPLET = 92
    integer, parameter, public :: HADRON_REMNANT_OCTET = 93
```

Generic particles for internal use in event analysis:

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: PRT_ANY = 81
    integer, parameter, public :: PRT_VISIBLE = 82
    integer, parameter, public :: PRT_CHARGED = 83
    integer, parameter, public :: PRT_COLORED = 84
```

Further particle codes for internal use:

⟨*Models: types*⟩+≡
```
    integer, parameter, public :: INVALID = 97
    integer, parameter, public :: KEYSTONE = 98
    integer, parameter, public :: COMPOSITE = 99
```

6.1.3 Spin codes

Somewhat redundant, but for better readability we define named constants for spin types. If the mass is nonzero, this is equal to the number of degrees of freedom.

⟨*Models: types*⟩+≡
```
    integer, parameter, public:: UNKNOWN=0
    integer, parameter, public :: SCALAR=1, SPINOR=2, VECTOR=3, &
         VECTORSPINOR=4, TENSOR=5
```

Isospin types and charge types are counted in an analogous way, where charge type 1 is charge 0, 2 is charge 1/3, and so on. Zero always means unknown. Note that charge and isospin types have an explicit sign.

Color types are defined as the dimension of the representation.

6.1.4 Particle data

The particle-data type holds all information that pertains to a particular particle within a particular model. Information such as spin type, particle code etc. is stored within the object itself, while mass and width are associated to parameters, otherwise assumed zero.

⟨*Models: public*⟩+≡
```
    public :: particle_data_t
```

⟨Models: types⟩+≡
```
    type :: particle_data_t
       private
       type(string_t) :: longname
       integer :: pdg = UNDEFINED
       logical :: is_visible = .true.
       logical :: is_parton = .false.
       logical :: is_gauge = .false.
       logical :: is_left_handed = .false.
       logical :: is_right_handed = .false.
       logical :: has_antiparticle = .false.
       logical :: p_is_stable = .true.
       logical :: p_decays_isotropically = .false.
       logical :: p_decays_diagonal = .false.
       logical :: a_is_stable = .true.
       logical :: a_decays_isotropically = .false.
       logical :: a_decays_diagonal = .false.
       logical :: p_polarized = .false.
       logical :: a_polarized = .false.
       type(string_t), dimension(:), allocatable :: name, anti
       type(string_t) :: tex_name, tex_anti
       integer :: spin_type = UNDEFINED
       integer :: isospin_type = 1
       integer :: charge_type = 1
       integer :: color_type = 1
       real(default), pointer :: mass_val => null ()
       type(parameter_t), pointer :: mass_src => null ()
       real(default), pointer :: width_val => null ()
       type(parameter_t), pointer :: width_src => null ()
       integer :: multiplicity = 1
       type(string_t), dimension(:), allocatable :: p_decay
       type(string_t), dimension(:), allocatable :: a_decay
    end type particle_data_t
```

Initialize particle data with PDG long name and PDG code. TeX names should
be initialized to avoid issues with accessing unallocated string contents.

⟨Models: public⟩+≡
```
    public :: particle_data_init
```

⟨Models: procedures⟩+≡
```
    subroutine particle_data_init (prt, longname, pdg)
      type(particle_data_t), intent(out) :: prt
      type(string_t), intent(in) :: longname
      integer, intent(in) :: pdg
      prt%longname = longname
      prt%pdg = pdg
      prt%tex_name = ""
      prt%tex_anti = ""
    end subroutine particle_data_init
```

Copy quantum numbers from another particle. Do not compute the multiplicity
yet, because this depends on the association of the `mass_src` pointer.

⟨Models: procedures⟩+≡

```
subroutine particle_data_copy (prt, prt_src)
  type(particle_data_t), intent(inout) :: prt
  type(particle_data_t), intent(in) :: prt_src
  prt%is_visible = prt_src%is_visible
  prt%is_parton = prt_src%is_parton
  prt%is_gauge = prt_src%is_gauge
  prt%is_left_handed = prt_src%is_left_handed
  prt%is_right_handed = prt_src%is_right_handed
  prt%p_is_stable =            prt_src%p_is_stable
  prt%p_decays_isotropically = prt_src%p_decays_isotropically
  prt%p_decays_diagonal =      prt_src%p_decays_diagonal
  prt%a_is_stable =            prt_src%a_is_stable
  prt%a_decays_isotropically = prt_src%a_decays_isotropically
  prt%a_decays_diagonal =      prt_src%a_decays_diagonal
  prt%p_polarized =            prt_src%p_polarized
  prt%a_polarized =            prt_src%a_polarized
  prt%spin_type = prt_src%spin_type
  prt%isospin_type = prt_src%isospin_type
  prt%charge_type = prt_src%charge_type
  prt%color_type = prt_src%color_type
  prt%has_antiparticle = prt_src%has_antiparticle
  if (allocated (prt_src%name)) then
     if (allocated (prt%name))  deallocate (prt%name)
     allocate (prt%name (size (prt_src%name)), source = prt_src%name)
  end if
  if (allocated (prt_src%anti)) then
     if (allocated (prt%anti))  deallocate (prt%anti)
     allocate (prt%anti (size (prt_src%anti)), source = prt_src%anti)
  end if
  prt%tex_name = prt_src%tex_name
  prt%tex_anti = prt_src%tex_anti
  if (allocated (prt_src%p_decay)) then
     if (allocated (prt%p_decay))  deallocate (prt%p_decay)
     allocate (prt%p_decay (size (prt_src%p_decay)), source = prt_src%p_decay)
  end if
  if (allocated (prt_src%a_decay)) then
     if (allocated (prt%a_decay))  deallocate (prt%a_decay)
     allocate (prt%a_decay (size (prt_src%a_decay)), source = prt_src%a_decay)
  end if
end subroutine particle_data_copy
```

Set particle quantum numbers.

⟨*Models: public*⟩+≡
```
    public :: particle_data_set
```

⟨*Models: procedures*⟩+≡
```
    subroutine particle_data_set (prt, &
        is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
        p_is_stable, p_decays_isotropically, p_decays_diagonal, &
        a_is_stable, a_decays_isotropically, a_decays_diagonal, &
        p_polarized, a_polarized, &
        name, anti, tex_name, tex_anti, &
        spin_type, isospin_type, charge_type, color_type, &
        mass_src, width_src, &
```

```fortran
       p_decay, a_decay)
  type(particle_data_t), intent(inout) :: prt
  logical, intent(in), optional :: is_visible, is_parton, is_gauge
  logical, intent(in), optional :: is_left_handed, is_right_handed
  logical, intent(in), optional :: p_is_stable
  logical, intent(in), optional :: p_decays_isotropically, p_decays_diagonal
  logical, intent(in), optional :: a_is_stable
  logical, intent(in), optional :: a_decays_isotropically, a_decays_diagonal
  logical, intent(in), optional :: p_polarized, a_polarized
  type(string_t), dimension(:), intent(in), optional :: name, anti
  type(string_t), intent(in), optional :: tex_name, tex_anti
  integer, intent(in), optional :: spin_type, isospin_type
  integer, intent(in), optional :: charge_type, color_type
  type(parameter_t), intent(in), pointer, optional :: mass_src, width_src
  type(string_t), dimension(:), intent(in), optional :: p_decay, a_decay
  if (present (is_visible))  prt%is_visible = is_visible
  if (present (is_parton))  prt%is_parton = is_parton
  if (present (is_gauge))  prt%is_gauge = is_gauge
  if (present (is_left_handed))  prt%is_left_handed = is_left_handed
  if (present (is_right_handed))  prt%is_right_handed = is_right_handed
  if (present (p_is_stable))  prt%p_is_stable = p_is_stable
  if (present (p_decays_isotropically)) &
       prt%p_decays_isotropically = p_decays_isotropically
  if (present (p_decays_diagonal)) &
       prt%p_decays_diagonal = p_decays_diagonal
  if (present (a_is_stable))  prt%a_is_stable = a_is_stable
  if (present (a_decays_isotropically)) &
       prt%a_decays_isotropically = a_decays_isotropically
  if (present (a_decays_diagonal)) &
       prt%a_decays_diagonal = a_decays_diagonal
  if (present (p_polarized)) prt%p_polarized = p_polarized
  if (present (a_polarized)) prt%a_polarized = a_polarized
  if (present (name)) then
     if (allocated (prt%name))  deallocate (prt%name)
     allocate (prt%name (size (name)), source = name)
  end if
  if (present (anti)) then
     if (allocated (prt%anti))  deallocate (prt%anti)
     allocate (prt%anti (size (anti)), source = anti)
     prt%has_antiparticle = .true.
  end if
  if (present (tex_name))  prt%tex_name = tex_name
  if (present (tex_anti))  prt%tex_anti = tex_anti
  if (present (spin_type))  prt%spin_type = spin_type
  if (present (isospin_type))  prt%isospin_type = isospin_type
  if (present (charge_type))  prt%charge_type = charge_type
  if (present (color_type))  prt%color_type = color_type
  if (present (mass_src)) then
     prt%mass_src => mass_src
     if (associated (mass_src)) then
        prt%mass_val => parameter_get_value_ptr (mass_src)
     else
        prt%mass_val => null ()
     end if
```

```
        end if
        if (present (width_src)) then
           prt%width_src => width_src
           if (associated (width_src)) then
              prt%width_val => parameter_get_value_ptr (width_src)
           else
              prt%width_val => null ()
           end if
        end if
        if (present (spin_type) .or. present (mass_src)) then
           call particle_data_set_multiplicity (prt)
        end if
        if (present (p_decay)) then
           if (allocated (prt%p_decay))  deallocate (prt%p_decay)
           if (size (p_decay) > 0) &
                allocate (prt%p_decay (size (p_decay)), source = p_decay)
        end if
        if (present (a_decay)) then
           if (allocated (prt%a_decay))  deallocate (prt%a_decay)
           if (size (a_decay) > 0) &
                allocate (prt%a_decay (size (a_decay)), source = a_decay)
        end if
      end subroutine particle_data_set
```

Calculate the multiplicity given spin type and mass.

⟨*Models: procedures*⟩+≡

```
      subroutine particle_data_set_multiplicity (prt)
        type(particle_data_t), intent(inout) :: prt
        if (prt%spin_type /= SCALAR) then
           if (associated (prt%mass_src)) then
              prt%multiplicity = prt%spin_type
           else if (prt%is_left_handed .or. prt%is_right_handed) then
              prt%multiplicity = 1
           else
              prt%multiplicity = 2
           end if
        end if
      end subroutine particle_data_set_multiplicity
```

Set the mass/width value (not the pointer). The mass/width pointer must be allocated.

⟨*Models: procedures*⟩+≡

```
      subroutine particle_data_set_mass (prt, mass)
        type(particle_data_t), intent(inout) :: prt
        real(default), intent(in) :: mass
        if (associated (prt%mass_val))  prt%mass_val = mass
      end subroutine particle_data_set_mass

      subroutine particle_data_set_width (prt, width)
        type(particle_data_t), intent(inout) :: prt
        real(default), intent(in) :: width
        if (associated (prt%width_val))  prt%width_val = width
      end subroutine particle_data_set_width
```

Loose ends

⟨*Models: public*⟩+≡

```
    public :: particle_data_freeze
```

⟨*Models: procedures*⟩+≡

```
    subroutine particle_data_freeze (prt)
      type(particle_data_t), intent(inout) :: prt
      if (.not. allocated (prt%name))  allocate (prt%name (0))
      if (.not. allocated (prt%anti))  allocate (prt%anti (0))
    end subroutine particle_data_freeze
```

Output

⟨*Models: procedures*⟩+≡

```
    subroutine particle_data_write (prt, unit)
      type(particle_data_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(3x,A,1x,A)", advance="no") "particle", char (prt%longname)
      write (u, "(1x,I0)", advance="no") prt%pdg
      if (.not. prt%is_visible) write (u, "(2x,A)", advance="no") "invisible"
      if (prt%is_parton)  write (u, "(2x,A)", advance="no") "parton"
      if (prt%is_gauge)   write (u, "(2x,A)", advance="no") "gauge"
      if (prt%is_left_handed)  write (u, "(2x,A)", advance="no") "left"
      if (prt%is_right_handed)  write (u, "(2x,A)", advance="no") "right"
      write (u, *)
      write (u, "(5x,A)", advance="no") "name"
      if (allocated (prt%name)) then
         do i = 1, size (prt%name)
            write (u, "(1x,A)", advance="no")  '"' // char (prt%name(i)) // '"'
         end do
         write (u, *)
         if (prt%has_antiparticle) then
            write (u, "(5x,A)", advance="no") "anti"
            do i = 1, size (prt%anti)
               write (u, "(1x,A)", advance="no")  '"' // char (prt%anti(i)) // '"'
            end do
            write (u, *)
         end if
         if (prt%tex_name /= "") then
            write (u, "(5x,A)")  &
                 "tex_name " // '"' // char (prt%tex_name) // '"'
         end if
         if (prt%has_antiparticle .and. prt%tex_anti /= "") then
            write (u, "(5x,A)")  &
                 "tex_anti " // '"' // char (prt%tex_anti) // '"'
         end if
      else
         write (u, "(A)")  "???"
      end if
      write (u, "(5x,A)", advance="no") "spin "
      select case (mod (prt%spin_type - 1, 2))
```

```
      case (0);  write (u, "(I0)", advance="no") (prt%spin_type-1) / 2
      case default;  write (u, "(I0,A)", advance="no") prt%spin_type-1, "/2"
      end select
      ! write (u, "(2x,A,I1,A)") "! [multiplicity = ", prt%multiplicity, "]"
      if (abs (prt%isospin_type) /= 1) then
         write (u, "(2x,A)", advance="no") "isospin "
         select case (mod (abs (prt%isospin_type) - 1, 2))
         case (0);  write (u, "(I0)", advance="no") &
               sign (abs (prt%isospin_type) - 1, prt%isospin_type) / 2
         case default;  write (u, "(I0,A)", advance="no") &
               sign (abs (prt%isospin_type) - 1, prt%isospin_type), "/2"
         end select
      end if
      if (abs (prt%charge_type) /= 1) then
         write (u, "(2x,A)", advance="no") "charge "
         select case (mod (abs (prt%charge_type) - 1, 3))
         case (0);  write (u, "(I0)", advance="no") &
               sign (abs (prt%charge_type) - 1, prt%charge_type) / 3
         case default;  write (u, "(I0,A)", advance="no") &
               sign (abs (prt%charge_type) - 1, prt%charge_type), "/3"
         end select
      end if
      if (prt%color_type /= 1) then
         write (u, "(2x,A,I0)", advance="no") "color ", prt%color_type
      end if
      write (u, *)
      if (associated (prt%mass_src)) then
         write (u, "(5x,A)", advance="no") "mass " // char (prt%mass_src%name)
         if (associated (prt%width_src)) then
            write (u, "(2x,A)") "width " // char (prt%width_src%name)
         else
            write (u, *)
         end if
      end if
      call particle_data_write_decays (prt, u)
    end subroutine particle_data_write
```

Write decay and polarization data.
⟨*Models: procedures*⟩+≡
```
    subroutine particle_data_write_decays (prt, unit)
      type(particle_data_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      if (.not. prt%p_is_stable) then
         if (allocated (prt%p_decay)) then
            write (u, "(5x,A)", advance="no") "p_decay"
            do i = 1, size (prt%p_decay)
               write (u, "(1x,A)", advance="no")  char (prt%p_decay(i))
            end do
            if (prt%p_decays_isotropically) then
               write (u, "(1x,A)", advance="no")  "isotropic"
            else if (prt%p_decays_diagonal) then
               write (u, "(1x,A)", advance="no")  "diagonal"
```

626

```
            end if
            write (u, *)
         end if
      else if (prt%p_polarized) then
         write (u, "(5x,A)")  "p_polarized"
      end if
      if (.not. prt%a_is_stable) then
         if (allocated (prt%a_decay)) then
            write (u, "(5x,A)", advance="no") "a_decay"
            do i = 1, size (prt%a_decay)
               write (u, "(1x,A)", advance="no")  char (prt%a_decay(i))
            end do
            if (prt%a_decays_isotropically) then
               write (u, "(1x,A)", advance="no")  "isotropic"
            else if (prt%a_decays_diagonal) then
               write (u, "(1x,A)", advance="no")  "diagonal"
            end if
            write (u, *)
         end if
      else if (prt%a_polarized) then
         write (u, "(5x,A)")  "a_polarized"
      end if
    end subroutine particle_data_write_decays
```

Screen version of output.
⟨*Models: procedures*⟩+≡

```
    subroutine particle_data_show (prt, l, u)
      type(particle_data_t), intent(in) :: prt
      integer, intent(in) :: l, u
      character(len=l) :: buffer
      integer :: i
      type(string_t), dimension(:), allocatable :: decay
      buffer = particle_data_get_name (prt, .false.)
      write (u, "(4x,A,1x,I8)", advance="no")  buffer, &
           particle_data_get_pdg (prt)
      if (particle_data_is_polarized (prt)) then
         write (u, "(3x,A)")  "polarized"
      else if (.not. particle_data_is_stable (prt)) then
         write (u, "(3x,A)", advance="no")  "decays:"
         call particle_data_get_decays (prt, decay)
         do i = 1, size (decay)
            write (u, "(1x,A)", advance="no")  char (decay(i))
         end do
         write (u, *)
      else
         write (u, *)
      end if
      if (particle_data_has_antiparticle (prt)) then
         buffer = particle_data_get_name (prt, .true.)
         write (u, "(4x,A,1x,I8)", advance="no")  buffer, &
              particle_data_get_pdg_anti (prt)
         if (particle_data_is_polarized (prt, .true.)) then
            write (u, "(3x,A)")  "polarized"
         else if (.not. particle_data_is_stable (prt, .true.)) then
```

627

```
              write (u, "(3x,A)", advance="no")  "decays:"
              call particle_data_get_decays (prt, decay, .true.)
              do i = 1, size (decay)
                 write (u, "(1x,A)", advance="no")  char (decay(i))
              end do
              write (u, *)
           else
              write (u, *)
           end if
        end if
     end subroutine particle_data_show
```

Retrieve data:

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_pdg
    public :: particle_data_get_pdg_anti
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_pdg (prt) result (pdg)
      integer :: pdg
      type(particle_data_t), intent(in) :: prt
      pdg = prt%pdg
    end function particle_data_get_pdg

    elemental function particle_data_get_pdg_anti (prt) result (pdg)
      integer :: pdg
      type(particle_data_t), intent(in) :: prt
      if (prt%has_antiparticle) then
         pdg = - prt%pdg
      else
         pdg = prt%pdg
      end if
    end function particle_data_get_pdg_anti
```

Predicates:

⟨*Models: public*⟩+≡
```
    public :: particle_data_is_visible
    public :: particle_data_is_parton
    public :: particle_data_is_gauge
    public :: particle_data_is_left_handed
    public :: particle_data_is_right_handed
    public :: particle_data_has_antiparticle
    public :: particle_data_is_stable
    public :: particle_data_get_decays
    public :: particle_data_decays_isotropically
    public :: particle_data_decays_diagonal
    public :: particle_data_is_polarized
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_is_visible (prt) result (flag)
      logical :: flag
      type(particle_data_t), intent(in) :: prt
      flag = prt%is_visible
    end function particle_data_is_visible
```

```
elemental function particle_data_is_parton (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_parton
end function particle_data_is_parton

elemental function particle_data_is_gauge (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_gauge
end function particle_data_is_gauge

elemental function particle_data_is_left_handed (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_left_handed
end function particle_data_is_left_handed

elemental function particle_data_is_right_handed (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%is_right_handed
end function particle_data_is_right_handed

elemental function particle_data_has_antiparticle (prt) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  flag = prt%has_antiparticle
end function particle_data_has_antiparticle

elemental function particle_data_is_stable (prt, anti) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        flag = prt%a_is_stable
     else
        flag = prt%p_is_stable
     end if
  else
     flag = prt%p_is_stable
  end if
end function particle_data_is_stable

subroutine particle_data_get_decays (prt, decay, anti)
  type(particle_data_t), intent(in) :: prt
  type(string_t), dimension(:), intent(out), allocatable :: decay
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        allocate (decay (size (prt%a_decay)), source = prt%a_decay)
     else
```

```
         allocate (decay (size (prt%p_decay)), source = prt%p_decay)
      end if
   else
      allocate (decay (size (prt%p_decay)), source = prt%p_decay)
   end if
end subroutine particle_data_get_decays

elemental function particle_data_decays_isotropically &
     (prt, anti) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        flag = prt%a_decays_isotropically
     else
        flag = prt%p_decays_isotropically
     end if
  else
     flag = prt%p_decays_isotropically
  end if
end function particle_data_decays_isotropically

elemental function particle_data_decays_diagonal &
     (prt, anti) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        flag = prt%a_decays_diagonal
     else
        flag = prt%p_decays_diagonal
     end if
  else
     flag = prt%p_decays_diagonal
  end if
end function particle_data_decays_diagonal

elemental function particle_data_is_polarized (prt, anti) result (flag)
  logical :: flag
  type(particle_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  logical :: a
  if (present (anti)) then
     a = anti
  else
     a = .false.
  end if
  if (a) then
     flag = prt%a_polarized
  else
     flag = prt%p_polarized
  end if
```

```
    end function particle_data_is_polarized
```

Names. Return the first name in the list (or the first antiparticle name)

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_name
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_name &
        (prt, is_antiparticle) result (name)
      type(string_t) :: name
      type(particle_data_t), intent(in) :: prt
      logical, intent(in) :: is_antiparticle
      name = prt%longname
      if (is_antiparticle) then
         if (prt%has_antiparticle) then
            if (allocated (prt%anti)) then
               if (size(prt%anti) > 0) name = prt%anti(1)
            end if
         else
            if (allocated (prt%name)) then
               if (size (prt%name) > 0) name = prt%name(1)
            end if
         end if
      else
         if (allocated (prt%name)) then
            if (size (prt%name) > 0) name = prt%name(1)
         end if
      end if
    end function particle_data_get_name
```

Same for the TEX name.

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_tex_name
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_tex_name &
        (prt, is_antiparticle) result (name)
      type(string_t) :: name
      type(particle_data_t), intent(in) :: prt
      logical, intent(in) :: is_antiparticle
      if (is_antiparticle) then
         if (prt%has_antiparticle) then
            name = prt%tex_anti
         else
            name = prt%tex_name
         end if
      else
         name = prt%tex_name
      end if
      if (name == "")  name = particle_data_get_name (prt, is_antiparticle)
    end function particle_data_get_tex_name
```

631

Quantum numbers

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_spin_type
    public :: particle_data_get_multiplicity
    public :: particle_data_get_isospin_type
    public :: particle_data_get_charge_type
    public :: particle_data_get_color_type
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_spin_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%spin_type
    end function particle_data_get_spin_type

    elemental function particle_data_get_multiplicity (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%multiplicity
    end function particle_data_get_multiplicity

    elemental function particle_data_get_isospin_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%isospin_type
    end function particle_data_get_isospin_type

    elemental function particle_data_get_charge_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%charge_type
    end function particle_data_get_charge_type

    elemental function particle_data_get_color_type (prt) result (type)
      integer :: type
      type(particle_data_t), intent(in) :: prt
      type = prt%color_type
    end function particle_data_get_color_type
```

In the MSSM, neutralinos can have a negative mass. This is relevant for computing matrix elements. However, within the WHIZARD main program we are interested only in kinematics, therefore we return the absolute value of the particle mass. If desired, we can extract the sign separately.

⟨*Models: public*⟩+≡
```
    public :: particle_data_get_charge
    public :: particle_data_get_mass
    public :: particle_data_get_mass_sign
    public :: particle_data_get_width
    public :: particle_data_get_isospin
```

⟨*Models: procedures*⟩+≡
```
    elemental function particle_data_get_charge (prt) result (charge)
      real(default) :: charge
      type(particle_data_t), intent(in) :: prt
```

```
      if (prt%charge_type /= 0) then
         charge = real (sign ((abs(prt%charge_type) - 1), &
               prt%charge_type), default) / 3
      else
         charge = 0
      end if
   end function particle_data_get_charge

   elemental function particle_data_get_mass (prt) result (mass)
     real(default) :: mass
     type(particle_data_t), intent(in) :: prt
     if (associated (prt%mass_val)) then
        mass = abs (prt%mass_val)
     else
        mass = 0
     end if
   end function particle_data_get_mass

   elemental function particle_data_get_mass_sign (prt) result (sgn)
     integer :: sgn
     type(particle_data_t), intent(in) :: prt
     if (associated (prt%mass_val)) then
        sgn = sign (1._default, prt%mass_val)
     else
        sgn = 0
     end if
   end function particle_data_get_mass_sign

   elemental function particle_data_get_width (prt) result (width)
     real(default) :: width
     type(particle_data_t), intent(in) :: prt
     if (associated (prt%width_val)) then
        width = prt%width_val
     else
        width = 0
     end if
   end function particle_data_get_width

   elemental function particle_data_get_isospin (prt) result (isospin)
     real(default) :: isospin
     type(particle_data_t), intent(in) :: prt
     if (prt%isospin_type /= 0) then
        isospin = real (sign (abs(prt%isospin_type) - 1, &
              prt%isospin_type), default) / 2
     else
        isospin = 0
     end if
   end function particle_data_get_isospin
```

Given an arrray of particles, return a PDG-array object that consists of all charged particles in the array. We need to explicitly add their antiparticles.

⟨*Models: procedures*⟩+≡
```
   function particle_data_get_charged_pdg (prt) result (aval)
```

633

```
      type(pdg_array_t) :: aval, aval_p, aval_a
      type(particle_data_t), dimension(:), intent(in) :: prt
      aval_p = pack ( prt%pdg, abs (prt%charge_type) > 1)
      aval_a = pack (-prt%pdg, abs (prt%charge_type) > 1 &
            .and. prt%has_antiparticle )
      aval = aval_p // aval_a
    end function particle_data_get_charged_pdg
```

The same for color.

⟨*Models: procedures*⟩+≡
```
    function particle_data_get_colored_pdg (prt) result (aval)
      type(pdg_array_t) :: aval, aval_p, aval_a
      type(particle_data_t), dimension(:), intent(in) :: prt
      aval_p = pack ( prt%pdg, abs (prt%color_type) > 1)
      aval_a = pack (-prt%pdg, abs (prt%color_type) > 1 &
            .and. prt%has_antiparticle )
      aval = aval_p // aval_a
    end function particle_data_get_colored_pdg
```

6.1.5 Vertex data

The vertex object contains an array of particle-data pointers, for which we need
a separate type. (We could use the flavor type defined in another module.)

   The program does not (yet?) make use of vertex definitions, so they are not
stored here.

⟨*Models: types*⟩+≡
```
    type :: particle_p
       type(particle_data_t), pointer :: p => null ()
    end type particle_p
```

⟨*Models: types*⟩+≡
```
    type :: vertex_t
       logical :: trilinear
       integer, dimension(:), allocatable :: pdg
       type(particle_p), dimension(:), allocatable :: prt
    end type vertex_t
```

Initialize using PDG codes. The model is used for finding particle data pointers
associated with the pdg codes.

⟨*Models: procedures*⟩+≡
```
    subroutine vertex_init (vtx, pdg, model)
      type(vertex_t), intent(out) :: vtx
      integer, dimension(:), intent(in) :: pdg
      type(model_t), intent(in), target, optional :: model
      integer :: i
      allocate (vtx%pdg (size (pdg)))
      allocate (vtx%prt (size (pdg)))
      vtx%trilinear = size (pdg) == 3
      vtx%pdg = pdg
      if (present (model)) then
```

```
            do i = 1, size (pdg)
               vtx%prt(i)%p => model_get_particle_ptr (model, pdg(i))
            end do
         end if
      end subroutine vertex_init
```

⟨*Models: procedures*⟩+≡
```
      subroutine vertex_write (vtx, unit)
        type(vertex_t), intent(in) :: vtx
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit);  if (u < 0)  return
        write (u, "(3x,A)", advance="no")  "vertex"
        do i = 1, size (vtx%prt)
           if (associated (vtx%prt(i)%p)) then
              write (u, "(1x,A)", advance="no") &
                   '"' // char (particle_data_get_name &
                                  (vtx%prt(i)%p, vtx%pdg(i) < 0)) &
                      // '"'
           else
              write (u, "(1x,I7)", advance="no") vtx%pdg(i)
           end if
        end do
        write (u, *)
      end subroutine vertex_write
```

Single-particle lookup: Given a particle code, we return matching codes if
present, otherwise zero. Actually, we return the antiparticles of the matching
codes, as appropriate for computing splittings.

  If the `radiative` flag is not set, we eliminate radiative decays. These have
the properties that the initial particle coincides with one of the outgoing parti-
cles.

⟨*Models: procedures*⟩+≡
```
      subroutine vertex_get_match (vtx, pdg1, pdg2, radiative)
        type(vertex_t), intent(in) :: vtx
        integer, intent(in) :: pdg1
        integer, dimension(:), allocatable, intent(out) :: pdg2
        logical, intent(in) :: radiative
        integer :: i, j
        do i = 1, size (vtx%pdg)
           if (vtx%pdg(i) == pdg1) then
              allocate (pdg2 (size (vtx%pdg) - 1))
              do j = 1, i-1
                 pdg2(j) = anti (j)
              end do
              do j = i, size (pdg2)
                 pdg2(j) = anti (j+1)
              end do
              exit
           end if
        end do
        if (allocated (pdg2)) then
           if (.not. radiative .and. any (pdg2 == pdg1))  deallocate (pdg2)
```

```
        end if
contains
  function anti (i) result (pdg)
    integer, intent(in) :: i
    integer :: pdg
    if (vtx%prt(i)%p%has_antiparticle) then
       pdg = - vtx%pdg(i)
    else
       pdg = vtx%pdg(i)
    end if
  end function anti
end subroutine vertex_get_match
```

6.1.6 Vertex lookup table

The vertex lookup table is a hash table: given two particle codes, we check
which codes are allowed for the third one.

The size of the hash table should be large enough that collisions are rare. We
first select a size based on the number of vertices (multiplied by six because all
permutations count), with some margin, and then choose the smallest integer
power of two larger than this.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: VERTEX_TABLE_SCALE_FACTOR = 60
```

⟨*Models: procedures*⟩+≡
```
    function vertex_table_size (n_vtx) result (n)
      integer(i32) :: n
      integer, intent(in) :: n_vtx
      integer :: i, s
      s = VERTEX_TABLE_SCALE_FACTOR * n_vtx
      n = 1
      do i = 1, 31
         n = ishft (n, 1)
         s = ishft (s,-1)
         if (s == 0)  exit
      end do
    end function vertex_table_size
```

The specific hash function takes two particle codes (arbitrary integers) and
returns a 32-bit integer. It makes use of the universal function `hash` which
operates on a byte array.

⟨*Models: procedures*⟩+≡
```
    function hash2 (pdg1, pdg2)
      integer(i32) :: hash2
      integer, intent(in) :: pdg1, pdg2
      integer(i8), dimension(1) :: mold
      hash2 = hash (transfer ((/pdg1, pdg2/), mold))
    end function hash2
```

Each entry in the vertex table stores the two particle codes and an array of possibilities for the third code.

⟨*Models: types*⟩+≡
```
type :: vertex_table_entry_t
   integer :: pdg1 = 0, pdg2 = 0
   integer :: n = 0
   integer, dimension(:), allocatable :: pdg3
end type vertex_table_entry_t
```

The vertex table:

⟨*Models: types*⟩+≡
```
type :: vertex_table_t
   type(vertex_table_entry_t), dimension(:), allocatable :: entry
   integer :: n_collisions = 0
   integer(i32) :: mask
end type vertex_table_t
```

Initializing the vertex table: This is done in two passes. First, we scan all permutations for all vertices and count the number of entries in each bucket of the hashtable. Then, the buckets are allocated accordingly and filled.

Collision resolution is done by just incrementing the hash value until an empty bucket is found. The vertex table size is fixed, since we know from the beginning the number of entries.

⟨*Models: procedures*⟩+≡
```
subroutine vertex_table_init (vt, prt, vtx)
  type(vertex_table_t), intent(out) :: vt
  type(particle_data_t), dimension(:), intent(in) :: prt
  type(vertex_t), dimension(:), intent(in) :: vtx
  integer :: n_vtx, vt_size, i, p1, p2, p3
  integer, dimension(3) :: p
  n_vtx = size (vtx)
  vt_size = vertex_table_size (count (vtx%trilinear))
  vt%mask = vt_size - 1
  allocate (vt%entry (0:vt_size-1))
  do i = 1, n_vtx
     if (vtx(i)%trilinear) then
        p = vtx(i)%pdg
        p1 = p(1);   p2 = p(2)
        call create (hash2 (p1, p2))
        if (p(2) /= p(3)) then
           p2 = p(3)
           call create (hash2 (p1, p2))
        end if
        if (p(1) /= p(2)) then
           p1 = p(2);   p2 = p(1)
           call create (hash2 (p1, p2))
           if (p(1) /= p(3)) then
              p2 = p(3)
              call create (hash2 (p1, p2))
           end if
        end if
        if (p(1) /= p(3)) then
```

```
                    p1 = p(3);   p2 = p(1)
                    call create (hash2 (p1, p2))
                    if (p(1) /= p(2)) then
                       p2 = p(2)
                       call create (hash2 (p1, p2))
                    end if
                 end if
              end if
         end do
         do i = 0, vt_size - 1
            allocate (vt%entry(i)%pdg3 (vt%entry(i)%n))
         end do
         vt%entry%n = 0
         do i = 1, n_vtx
            if (vtx(i)%trilinear) then
               p = vtx(i)%pdg
               p1 = p(1);   p2 = p(2);   p3 = p(3)
               call register (hash2 (p1, p2))
               if (p(2) /= p(3)) then
                  p2 = p(3);   p3 = p(2)
                  call register (hash2 (p1, p2))
               end if
               if (p(1) /= p(2)) then
                  p1 = p(2);   p2 = p(1);   p3 = p(3)
                  call register (hash2 (p1, p2))
                  if (p(1) /= p(3)) then
                     p2 = p(3);   p3 = p(1)
                     call register (hash2 (p1, p2))
                  end if
               end if
               if (p(1) /= p(3)) then
                  p1 = p(3);   p2 = p(1);   p3 = p(2)
                  call register (hash2 (p1, p2))
                  if (p(1) /= p(2)) then
                     p2 = p(2);   p3 = p(1)
                     call register (hash2 (p1, p2))
                  end if
               end if
            end if
         end do
      contains
        recursive subroutine create (hashval)
          integer(i32), intent(in) :: hashval
          integer :: h
          h = iand (hashval, vt%mask)
          if (vt%entry(h)%n == 0) then
             vt%entry(h)%pdg1 = p1
             vt%entry(h)%pdg2 = p2
             vt%entry(h)%n = 1
          else if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
             vt%entry(h)%n = vt%entry(h)%n + 1
          else
             vt%n_collisions = vt%n_collisions + 1
             call create (hashval + 1)
```

```
          end if
      end subroutine create
      recursive subroutine register (hashval)
        integer(i32), intent(in) :: hashval
        integer :: h
        h = iand (hashval, vt%mask)
        if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
           vt%entry(h)%n = vt%entry(h)%n + 1
           vt%entry(h)%pdg3(vt%entry(h)%n) = p3
        else
           call register (hashval + 1)
        end if
      end subroutine register
  end subroutine vertex_table_init
```

Output

⟨*Models: procedures*⟩+≡

```
  subroutine vertex_table_write (vt, unit)
    type(vertex_table_t), intent(in) :: vt
    integer, intent(in), optional :: unit
    integer :: u, i
    character(9) :: size_pdg3
    u = output_unit (unit);  if (u < 0)  return
    write (u, "(A)") "vertex hash table:"
    write (u, "(A,I7)") "  size = ", size (vt%entry)
    write (u, "(A,I7)") "  used = ", count (vt%entry%n /= 0)
    write (u, "(A,I7)") "  coll = ", vt%n_collisions
    do i = lbound (vt%entry, 1), ubound (vt%entry, 1)
       if (vt%entry(i)%n /= 0) then
          write (size_pdg3, "(I7)") size (vt%entry(i)%pdg3)
          write (u, "(A,1x,I7,1x,A,2(1x,I7),A," // &
               size_pdg3 // "(1x,I7))")  &
               " ", i, ":", vt%entry(i)%pdg1, &
               vt%entry(i)%pdg2, "->", vt%entry(i)%pdg3
       end if
    end do
  end subroutine vertex_table_write
```

Return the array of particle codes that match the given pair.

⟨*Models: procedures*⟩+≡

```
  subroutine vertex_table_match (vt, pdg1, pdg2, pdg3)
    type(vertex_table_t), intent(in) :: vt
    integer, intent(in) :: pdg1, pdg2
    integer, dimension(:), allocatable, intent(out) :: pdg3
    call match (hash2 (pdg1, pdg2))
  contains
    recursive subroutine match (hashval)
      integer(i32), intent(in) :: hashval
      integer :: h
      h = iand (hashval, vt%mask)
      if (vt%entry(h)%n == 0) then
         allocate (pdg3 (0))
      else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
```

```
        allocate (pdg3 (size (vt%entry(h)%pdg3)))
        pdg3 = vt%entry(h)%pdg3
      else
        call match (hashval + 1)
      end if
    end subroutine match
  end subroutine vertex_table_match
```

Return true if the triplet is represented as a vertex.

⟨*Models: procedures*⟩+≡
```
  function vertex_table_check (vt, pdg1, pdg2, pdg3) result (flag)
    type(vertex_table_t), intent(in) :: vt
    integer, intent(in) :: pdg1, pdg2, pdg3
    logical :: flag
    flag = check (hash2 (pdg1, pdg2))
  contains
    recursive function check (hashval) result (flag)
      integer(i32), intent(in) :: hashval
      integer :: h
      logical :: flag
      h = iand (hashval, vt%mask)
      if (vt%entry(h)%n == 0) then
         flag = .false.
      else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
         flag = any (vt%entry(h)%pdg3 == pdg3)
      else
         flag = check (hashval + 1)
      end if
    end function check
  end function vertex_table_check
```

6.1.7   Model data

A model object holds all information about parameters, particles, and vertices. For models that require an external program for parameter calculation, there is the pointer to a function that does this calculation, given the set of independent and derived parameters.

When reading a model, we create a parse tree. Parameter definitions are available via parse nodes. Since we may need those later when making model instances, we keep the whole parse tree in the model definition (but not in the instances).

⟨*Models: public*⟩+≡
```
  public :: model_t
```

⟨*Models: types*⟩+≡
```
  type :: model_t
    private
    type(string_t) :: name
    character(32) :: md5sum = ""
    type(parameter_t), dimension(:), allocatable :: par
    type(particle_data_t), dimension(:), allocatable :: prt
    type(vertex_t), dimension(:), allocatable :: vtx
```

```
    type(vertex_table_t) :: vt
    type(var_list_t) :: var_list
    type(string_t) :: dlname
    procedure(model_init_external_parameters), nopass, pointer :: &
         init_external_parameters => null ()
    type(dlaccess_t) :: dlaccess
    type(parse_tree_t) :: parse_tree
 contains
⟨Models: model: TBP⟩
end type model_t
```

This is the interface for a procedure that initializes the calculation of external parameters, given the array of all parameters.

⟨*Models: interfaces*⟩≡
```
    abstract interface
      subroutine model_init_external_parameters (par) bind (C)
        import
        real(c_default_float), dimension(*), intent(inout) :: par
      end subroutine model_init_external_parameters
    end interface
```

Initialization: Specify the number of parameters, particles, vertices and allocate memory. If an associated DL library is specified, load this library.

⟨*Models: procedures*⟩+≡
```
    subroutine model_init &
         (model, name, libname, os_data, n_par, n_prt, n_vtx)
      type(model_t), intent(inout) :: model
      type(string_t), intent(in) :: name, libname
      type(os_data_t), intent(in) :: os_data
      integer, intent(in) :: n_par, n_prt, n_vtx
      type(c_funptr) :: c_fptr
      type(string_t) :: libpath
      call model_basic_init (model, name, n_par, n_prt, n_vtx)
      if (libname /= "") then
         if (.not. os_data%use_testfiles) then
            libpath = os_data%whizard_models_libpath_local
            model%dlname = os_get_dlname ( &
              libpath // "/" // libname, os_data, ignore=.true.)
         end if
         if (model%dlname == "") then
            libpath = os_data%whizard_models_libpath
            model%dlname = os_get_dlname (libpath // "/" // libname, os_data)
         end if
      else
         model%dlname = ""
      end if
      if (model%dlname /= "") then
         if (.not. dlaccess_is_open (model%dlaccess)) then
            if (logging) &
                 call msg_message ("Loading model auxiliary library '" &
                 // char (libpath) // "/" // char (model%dlname) // "'")
            call dlaccess_init (model%dlaccess, os_data%whizard_models_libpath, &
                 model%dlname, os_data)
```

```
            if (dlaccess_has_error (model%dlaccess)) then
               call msg_message (char (dlaccess_get_error (model%dlaccess)))
               call msg_fatal ("Loading model auxiliary library '" &
                    // char (model%dlname) // "' failed")
               return
            end if
            c_fptr = dlaccess_get_c_funptr (model%dlaccess, &
                 var_str ("init_external_parameters"))
            if (dlaccess_has_error (model%dlaccess)) then
               call msg_message (char (dlaccess_get_error (model%dlaccess)))
               call msg_fatal ("Loading function from auxiliary library '" &
                    // char (model%dlname) // "' failed")
               return
            end if
            call c_f_procpointer (c_fptr, model% init_external_parameters)
         end if
      end if
    end subroutine model_init
```

For a model instance, we do not attempt to load a DL library. This is the core of the full initializer above.

⟨*Models: procedures*⟩+≡
```
    subroutine model_basic_init (model, name, n_par, n_prt, n_vtx)
      type(model_t), intent(inout) :: model
      type(string_t), intent(in) :: name
      integer, intent(in) :: n_par, n_prt, n_vtx
      model%name = name
      allocate (model%par (n_par))
      allocate (model%prt (n_prt))
      allocate (model%vtx (n_vtx))
    end subroutine model_basic_init
```

Finalization: The variable list contains allocated pointers, also the parse tree. We also close the DL access object, if any, that enables external parameter calculation.

⟨*Models: public*⟩+≡
```
    public :: model_final
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_final (model)
      type(model_t), intent(inout) :: model
      integer :: i
      if (allocated (model%par)) then
         do i = 1, size (model%par)
            call parameter_final (model%par(i))
         end do
      end if
      call var_list_final (model%var_list)
      if (model%dlname /= "")  call dlaccess_final (model%dlaccess)
      call parse_tree_final (model%parse_tree)
    end subroutine model_final
```

Output. By default, the output is in the form of an input file. If `verbose` is true, for each derived parameter the definition (eval tree) is displayed, and the vertex hash table is shown.

⟨*Models: public*⟩+≡
```
public :: model_write
```

⟨*Models: model: TBP*⟩≡
```
procedure :: write => model_write
```

⟨*Models: procedures*⟩+≡
```
subroutine model_write (model, unit, verbose, &
    show_md5sum, show_variables, show_parameters, &
    show_particles, show_vertices)
  class(model_t), intent(in) :: model
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical, intent(in), optional :: show_md5sum
  logical, intent(in), optional :: show_variables
  logical, intent(in), optional :: show_parameters
  logical, intent(in), optional :: show_particles
  logical, intent(in), optional :: show_vertices
  logical :: verb, show_md5, show_var, show_par, show_prt, show_vtx
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  verb = .false.;  if (present (verbose))  verb = verbose
  show_md5 = .true.;  if (present (show_md5sum)) &
      show_md5 = show_md5sum
  show_par = .true.;  if (present (show_parameters)) &
      show_par = show_parameters
  show_prt = .true.;  if (present (show_particles)) &
      show_prt = show_particles
  show_vtx = .true.;  if (present (show_vertices)) &
      show_vtx = show_vertices
  show_var = verb;  if (present (show_variables)) &
      show_var = show_variables
  write (u, "(A,A,A)") 'model "', char (model%name), '"'
  if (show_md5 .and. model%md5sum /= "") &
      write (u, "(1x,A,A,A)") "! md5sum = '", model%md5sum, "'"
  if (show_par) then
     write (u, *)
     do i = 1, size (model%par)
        call parameter_write (model%par(i), unit, verbose)
     end do
  end if
  if (show_prt) then
     write (u, *)
     do i = 1, size (model%prt)
        call particle_data_write (model%prt(i), unit)
     end do
  end if
  if (show_vtx) then
     write (u, *)
     do i = 1, size (model%vtx)
        call vertex_write (model%vtx(i), unit)
     end do
```

```
               if (verb) then
                  write (u, *)
                  call vertex_table_write (model%vt, unit)
               end if
            end if
            if (show_var) then
               write (u, *)
               call var_list_write (model%var_list, unit)
            end if
         end subroutine model_write
```

Screen output, condensed form.

⟨*Models: public*⟩+≡
```
      public :: model_show
```

⟨*Models: model: TBP*⟩+≡
```
      procedure :: show => model_show
```

⟨*Models: procedures*⟩+≡
```
   subroutine model_show (model, unit)
     class(model_t), intent(in) :: model
     integer, intent(in), optional :: unit
     integer :: i, u, l
     u = output_unit (unit)
     write (u, "(A,1x,A)")  "Model:", char (model%name)
     l = max (maxval (len (particle_data_get_name (model%prt, .false.))), &
         maxval (len (particle_data_get_name (model%prt, .true.))))
     write (u, "(2x,A)")  "Particles:"
     do i = 1, size (model%prt)
        call particle_data_show (model%prt(i), l, u)
     end do
     l = maxval (len (model%par%name))
     if (any (model%par%type == PAR_INDEPENDENT)) then
        write (u, "(2x,A)")  "Independent parameters:"
        do i = 1, size (model%par)
           call parameter_show_independent (model%par(i), l, u)
        end do
     end if
     if (any (model%par%type == PAR_DERIVED)) then
        write (u, "(2x,A)")  "Derived parameters:"
        do i = 1, size (model%par)
           call parameter_show_derived (model%par(i), l, u)
        end do
     end if
     if (any (model%par%type == PAR_EXTERNAL)) then
        write (u, "(2x,A)")  "External parameters:"
        do i = 1, size (model%par)
           call parameter_show_external (model%par(i), l, u)
        end do
     end if
   end subroutine model_show
```

Show the list of stable, unstable, polarized, or unpolarized particles, respectively.

⟨*Models: public*⟩+≡

```
      public :: model_show_stable
      public :: model_show_unstable
      public :: model_show_polarized
      public :: model_show_unpolarized
⟨Models: procedures⟩+≡
      subroutine model_show_stable (model, unit)
        type(model_t), intent(in) :: model
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit)
        write (u, "(A,1x)", advance="no")  "Stable particles:"
        do i = 1, size (model%prt)
           associate (prt => model%prt(i))
             if (particle_data_is_stable (prt, .false.)) then
                write (u, "(1x,A)", advance="no") &
                     char (particle_data_get_name (prt, .false.))
             end if
             if (particle_data_has_antiparticle (prt)) then
                if (particle_data_is_stable (prt, .true.)) then
                   write (u, "(1x,A)", advance="no") &
                        char (particle_data_get_name (prt, .true.))
                end if
             end if
           end associate
        end do
        write (u, *)
      end subroutine model_show_stable

      subroutine model_show_unstable (model, unit)
        type(model_t), intent(in) :: model
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit)
        write (u, "(A,1x)", advance="no")  "Unstable particles:"
        do i = 1, size (model%prt)
           associate (prt => model%prt(i))
             if (.not. particle_data_is_stable (prt, .false.)) then
                write (u, "(1x,A)", advance="no") &
                     char (particle_data_get_name (prt, .false.))
             end if
             if (particle_data_has_antiparticle (prt)) then
                if (.not. particle_data_is_stable (prt, .true.)) then
                   write (u, "(1x,A)", advance="no") &
                        char (particle_data_get_name (prt, .true.))
                end if
             end if
           end associate
        end do
        write (u, *)
      end subroutine model_show_unstable

      subroutine model_show_polarized (model, unit)
        type(model_t), intent(in) :: model
        integer, intent(in), optional :: unit
```

```
        integer :: u, i
        u = output_unit (unit)
        write (u, "(A,1x)", advance="no")  "Polarized particles:"
        do i = 1, size (model%prt)
           associate (prt => model%prt(i))
             if (particle_data_is_polarized (prt, .false.)) then
                write (u, "(1x,A)", advance="no") &
                     char (particle_data_get_name (prt, .false.))
             end if
             if (particle_data_has_antiparticle (prt)) then
                if (particle_data_is_polarized (prt, .true.)) then
                   write (u, "(1x,A)", advance="no") &
                        char (particle_data_get_name (prt, .true.))
                end if
             end if
           end associate
        end do
        write (u, *)
      end subroutine model_show_polarized

      subroutine model_show_unpolarized (model, unit)
        type(model_t), intent(in) :: model
        integer, intent(in), optional :: unit
        integer :: u, i
        u = output_unit (unit)
        write (u, "(A,1x)", advance="no")  "Unpolarized particles:"
        do i = 1, size (model%prt)
           associate (prt => model%prt(i))
             if (.not. particle_data_is_polarized (prt, .false.)) then
                write (u, "(1x,A)", advance="no") &
                     char (particle_data_get_name (prt, .false.))
             end if
             if (particle_data_has_antiparticle (prt)) then
                if (.not. particle_data_is_polarized (prt, .true.)) then
                   write (u, "(1x,A)", advance="no") &
                        char (particle_data_get_name (prt, .true.))
                end if
             end if
           end associate
        end do
        write (u, *)
      end subroutine model_show_unpolarized
```

Accessing contents:

⟨*Models: public*⟩+≡
```
    public :: model_get_name
```

⟨*Models: model: TBP*⟩+≡
```
    procedure :: get_name => model_get_name
```

⟨*Models: procedures*⟩+≡
```
    function model_get_name (model) result (name)
      type(string_t) :: name
      class(model_t), intent(in) :: model
      name = model%name
```

```
      end function model_get_name
```

Retrieve the MD5 sum of a model (actually, of the model file).

⟨*Models: public*⟩+≡
```
    public :: model_get_md5sum
```

⟨*Models: procedures*⟩+≡
```
    function model_get_md5sum (model) result (md5sum)
      character(32) :: md5sum
      type(model_t), intent(in) :: model
      md5sum = model%md5sum
    end function model_get_md5sum
```

Retrieve a MD5 sum for the current model parameter values and decay/polarization settings. This is done by writing them to a temporary file, using a standard format.

⟨*Models: public*⟩+≡
```
    public :: model_get_parameters_md5sum
```

⟨*Models: procedures*⟩+≡
```
    function model_get_parameters_md5sum (model) result (par_md5sum)
      character(32) :: par_md5sum
      type(model_t), intent(in) :: model
      real(default), dimension(:), allocatable :: par
      integer :: unit, i
      call model_parameters_to_array (model, par)
      unit = free_unit ()
      open (unit, status="scratch", action="readwrite")
      write (unit, "(" // FMT_19 // ")")  par
      do i = 1, size (model%prt)
         associate (prt => model%prt(i))
           if (.not. prt%p_is_stable .or. .not. prt%a_is_stable &
                .or. prt%p_polarized .or. prt%a_polarized) then
             write (unit, "(3x,A)") char (prt%longname)
             call particle_data_write_decays (prt, unit)
           end if
         end associate
      end do
      rewind (unit)
      par_md5sum = md5sum (unit)
      close (unit)
    end function model_get_parameters_md5sum
```

Parameters are defined by an expression which may be constant or arbitrary.

⟨*Models: interfaces*⟩+≡
```
    interface model_set_parameter
       module procedure model_set_parameter_constant
       module procedure model_set_parameter_parse_node
    end interface
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_set_parameter_constant (model, i, name, value)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), intent(in) :: name
      real(default), intent(in) :: value
      logical, save, target :: known = .true.
      call parameter_init_independent_value (model%par(i), name, value)
      call var_list_append_real_ptr &
           (model%var_list, name, parameter_get_value_ptr (model%par(i)), known, &
           intrinsic=.true.)
    end subroutine model_set_parameter_constant

    subroutine model_set_parameter_parse_node (model, i, name, pn, constant)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), intent(in) :: name
      type(parse_node_t), intent(in), target :: pn
      logical, intent(in) :: constant
      logical, save, target :: known = .true.
      if (constant) then
         call parameter_init_independent (model%par(i), name, pn)
      else
         call parameter_init_derived (model%par(i), name, pn, model%var_list)
      end if
      call var_list_append_real_ptr &
           (model%var_list, name, parameter_get_value_ptr (model%par(i)), &
            is_known=known, locked=.not.constant, intrinsic=.true.)
    end subroutine model_set_parameter_parse_node

    subroutine model_set_parameter_external (model, i, name)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), intent(in) :: name
      logical, save, target :: known = .true.
      call parameter_init_external (model%par(i), name)
      call var_list_append_real_ptr &
           (model%var_list, name, parameter_get_value_ptr (model%par(i)), &
            is_known=known, locked=.true., intrinsic=.true.)
    end subroutine model_set_parameter_external
```

Make a copy of a parameter. The eval tree should not be copied, since it should refer to the new variable list. The safe solution is to make use of the above initializers, which also take care of the building a new variable list, but then copy the actual values.

⟨*Models: procedures*⟩+≡
```
    subroutine model_copy_parameter (model, i, par)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(parameter_t), intent(in) :: par
      select case (par%type)
      case (PAR_INDEPENDENT)
         if (associated (par%pn)) then
```

```
            call model_set_parameter_parse_node (model, i, par%name, par%pn, &
                 constant = .true.)
          else
             call model_set_parameter_constant (model, i, par%name, par%value)
          end if
       case (PAR_DERIVED)
          call model_set_parameter_parse_node (model, i, par%name, par%pn, &
               constant = .false.)
       case (PAR_EXTERNAL)
          call model_set_parameter_external (model, i, par%name)
       end select
       model%par(i)%value = par%value
     end subroutine model_copy_parameter
```

Return the pointer to a parameter

⟨*Models: procedures*⟩+≡

```
    function model_get_parameter_ptr (model, par_name) result (par)
      type(parameter_t), pointer :: par
      type(model_t), intent(in), target :: model
      type(string_t), intent(in), optional :: par_name
      integer :: i
      par => null ()
      if (present (par_name)) then
         do i = 1, size (model%par)
            if (model%par(i)%name == par_name) then
               par => model%par(i);  exit
            end if
         end do
         if (.not. associated (par)) then
            call msg_fatal (" Model '" // char (model%name) // "'" // &
                 " has no parameter '" // char (par_name) // "'")
         end if
      end if
    end function model_get_parameter_ptr
```

Return the value of a particular parameter

⟨*Models: public*⟩+≡

```
    public :: model_get_parameter_value
```

⟨*Models: procedures*⟩+≡

```
    function model_get_parameter_value (model, par_name) result (val)
      real(default) :: val
      type(model_t), intent(in), target :: model
      type(string_t), intent(in), optional :: par_name
      val = parameter_get_value_ptr (model_get_parameter_ptr (model, par_name))
    end function model_get_parameter_value
```

Return the number of parameters

⟨*Models: public*⟩+≡

```
    public :: model_get_n_parameters
```

⟨*Models: procedures*⟩+≡

```
    function model_get_n_parameters (model) result (n)
```

```
      integer :: n
      type(model_t), intent(in) :: model
      n = size (model%par)
    end function model_get_n_parameters
```

Transform the parameters into a single real-valued array. The first version uses
the Fortran default kind, the second one the C default kind. (Usually, the two
will be identical).

⟨*Models: public*⟩+≡
```
    public :: model_parameters_to_array
    public :: model_parameters_to_c_array
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_parameters_to_array (model, array)
      type(model_t), intent(in) :: model
      real(default), dimension(:), allocatable :: array
      integer :: i
      if (allocated (array))  deallocate (array)
      allocate (array (size (model%par)))
      do i = 1, size (model%par)
         array(i) = model%par(i)%value
      end do
    end subroutine model_parameters_to_array

    subroutine model_parameters_to_c_array (model, array)
      type(model_t), intent(in) :: model
      real(c_default_float), dimension(:), allocatable :: array
      allocate (array (size (model%par)))
      array = model%par%value
    end subroutine model_parameters_to_c_array

    subroutine model_parameters_from_c_array (model, array)
      type(model_t), intent(inout) :: model
      real(c_default_float), dimension(:), intent(in) :: array
      if (size (array) == size (model%par)) then
         model%par%value = array
      else
         call msg_bug ("Model '" // char (model%name) // "': size mismatch " &
              // "in parameter array")
      end if
    end subroutine model_parameters_from_c_array
```

Rcalculate all derived parameters.

⟨*Models: public*⟩+≡
```
    public :: model_parameters_update
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_parameters_update (model)
      type(model_t), intent(inout) :: model
      integer :: i
      real(default), dimension(:), allocatable :: par
      do i = 1, size (model%par)
         call parameter_reset_derived (model%par(i))
      end do
```

```
      if (associated (model%init_external_parameters)) then
         call model_parameters_to_c_array (model, par)
         call model%init_external_parameters (par)
         call model_parameters_from_c_array (model, par)
      end if
   end subroutine model_parameters_update
```

Initialize particle data with PDG long name and PDG code.

⟨*Models: procedures*⟩+≡
```
   subroutine model_init_particle (model, i, longname, pdg)
     type(model_t), intent(inout) :: model
     integer, intent(in) :: i
     type(string_t), intent(in) :: longname
     integer, intent(in) :: pdg
     type(pdg_array_t) :: aval
     call particle_data_init (model%prt(i), longname, pdg)
     aval = pdg
     call var_list_append_pdg_array &
          (model%var_list, longname, aval, locked=.true., intrinsic=.true.)
   end subroutine model_init_particle
```

Copy quantum numbers from another particle

⟨*Models: procedures*⟩+≡
```
   subroutine model_copy_particle_data (model, i, name_src)
     type(model_t), intent(inout), target :: model
     integer, intent(in) :: i
     type(string_t), intent(in) :: name_src
     call particle_data_copy (model%prt(i), &
          model_get_particle_ptr (model, &
             model_get_particle_pdg (model, name_src)))
   end subroutine model_copy_particle_data
```

Set particle quantum numbers individually. Names get a * prepended.

⟨*Models: procedures*⟩+≡
```
   subroutine model_set_particle_data (model, i, &
        is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
        name, anti, tex_name, tex_anti, &
        spin_type, isospin_type, charge_type, color_type, &
        mass_src, width_src)
     type(model_t), intent(inout) :: model
     integer, intent(in) :: i
     logical, intent(in), optional :: is_visible, is_parton, is_gauge
     logical, intent(in), optional :: is_left_handed, is_right_handed
     type(string_t), dimension(:), intent(in), optional :: name, anti
     type(string_t), intent(in), optional :: tex_name, tex_anti
     integer, intent(in), optional :: spin_type, isospin_type
     integer, intent(in), optional :: charge_type, color_type
     type(parameter_t), intent(in), optional, pointer :: mass_src, width_src
     integer :: j
     type(pdg_array_t) :: aval
     logical, parameter :: is_stable = .true.
     logical, parameter :: decays_isotropically = .false.
```

```
      logical, parameter :: decays_diagonal = .false.
      logical, parameter :: p_polarized = .false.
      logical, parameter :: a_polarized = .false.
      call particle_data_set (model%prt(i), &
         is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
         is_stable, decays_isotropically, decays_diagonal, &
         is_stable, decays_isotropically, decays_diagonal, &
         p_polarized, a_polarized, &
         name, anti, tex_name, tex_anti, &
         spin_type, isospin_type, charge_type, color_type, &
         mass_src, width_src)
      if (present (name)) then
         aval = particle_data_get_pdg (model%prt(i))
         do j = 1, size (name)
            call var_list_append_pdg_array &
                 (model%var_list, name(j), aval, locked=.true., intrinsic=.true.)
         end do
      end if
      if (present (anti)) then
         aval = - particle_data_get_pdg (model%prt(i))
         do j = 1, size (anti)
            call var_list_append_pdg_array &
                 (model%var_list, anti(j), aval, locked=.true., intrinsic=.true.)
         end do
      end if
    end subroutine model_set_particle_data
```

Mark a particle as unstable and provide a list of names for its decay processes.
In contrast with the previous subroutine which is for internal use, we address the
particle by its PDG code. If the index is negative, we address the antiparticle.

⟨*Models: public*⟩+≡
```
    public :: model_set_unstable
    public :: model_set_stable
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_set_unstable (model, pdg, decay, isotropic, diagonal)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      type(string_t), dimension(:), intent(in) :: decay
      logical, intent(in), optional :: isotropic, diagonal
      integer :: i
      i = model_get_particle_index (model, pdg)
      if (pdg > 0) then
         call particle_data_set (model%prt(i), &
              p_is_stable = .false., p_decay = decay, &
              p_decays_isotropically = isotropic, &
              p_decays_diagonal = diagonal)
      else
         call particle_data_set (model%prt(i), &
              a_is_stable = .false., a_decay = decay, &
              a_decays_isotropically = isotropic, &
              a_decays_diagonal = diagonal)
      end if
    end subroutine model_set_unstable
```

```
    subroutine model_set_stable (model, pdg)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      integer :: i
      i = model_get_particle_index (model, pdg)
      if (pdg > 0) then
         call particle_data_set (model%prt(i), &
              p_is_stable = .true.)
      else
         call particle_data_set (model%prt(i), &
              a_is_stable = .true.)
      end if
    end subroutine model_set_stable
```

Mark a particle as polarized.

⟨*Models: public*⟩+≡
```
    public :: model_set_polarized
    public :: model_set_unpolarized
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_set_polarized (model, pdg)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      integer :: i
      i = model_get_particle_index (model, pdg)
      if (pdg > 0) then
         call particle_data_set (model%prt(i), &
              p_polarized = .true.)
      else
         call particle_data_set (model%prt(i), &
              a_polarized = .true.)
      end if
    end subroutine model_set_polarized

    subroutine model_set_unpolarized (model, pdg)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      integer :: i
      i = model_get_particle_index (model, pdg)
      if (pdg > 0) then
         call particle_data_set (model%prt(i), &
              p_polarized = .false.)
      else
         call particle_data_set (model%prt(i), &
              a_polarized = .false.)
      end if
    end subroutine model_set_unpolarized
```

Revert all polarized (unstable) particles to unpolarized (stable) status, respectively.

⟨*Models: public*⟩+≡
```
    public :: model_clear_unstable
    public :: model_clear_polarized
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_clear_unstable (model)
      type(model_t), intent(inout) :: model
      integer :: i
      do i = 1, size (model%prt)
         associate (prt => model%prt(i))
           if (.not. particle_data_is_stable (prt, .false.)) then
              call particle_data_set (model%prt(i), p_is_stable = .true.)
           end if
           if (particle_data_has_antiparticle (prt)) then
              if (.not. particle_data_is_stable (prt, .true.)) then
                 call particle_data_set (model%prt(i), a_is_stable = .true.)
              end if
           end if
         end associate
      end do
    end subroutine model_clear_unstable

    subroutine model_clear_polarized (model)
      type(model_t), intent(inout) :: model
      integer :: i
      do i = 1, size (model%prt)
         associate (prt => model%prt(i))
           if (particle_data_is_polarized (prt, .false.)) then
              call particle_data_set (model%prt(i), p_polarized = .false.)
           end if
           if (particle_data_has_antiparticle (prt)) then
              if (particle_data_is_polarized (prt, .true.)) then
                 call particle_data_set (model%prt(i), a_polarized = .false.)
              end if
           end if
         end associate
      end do
    end subroutine model_clear_polarized
```

Finalize the particle data: allocate zero-sized arrays for particle name as defaults.

⟨*Models: procedures*⟩+≡
```
    subroutine model_freeze_particle_data (model, i)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: i
      call particle_data_freeze (model%prt(i))
    end subroutine model_freeze_particle_data
```

Return the total number of particles in the model.

⟨*Models: public*⟩+≡
```
    public :: model_get_n_particles
```

⟨*Models: procedures*⟩+≡
```
    function model_get_n_particles (model) result (n)
      type(model_t), intent(in) :: model
      integer :: n
      n = size (model%prt)
```

```
    end function model_get_n_particles
```

Find the index that belongs to a particle.

⟨Models: procedures⟩+≡
```
    function model_get_particle_index (model, pdg) result (index)
      type(model_t), intent(in) :: model
      integer, intent(in) :: pdg
      integer :: index
      integer :: i
      do i = 1, size (model%prt)
         if (model%prt(i)%pdg == abs (pdg)) then
            index = i;  return
         end if
      end do
      write (msg_buffer, "(1x,A,1x,I0)")  "PDG code =", pdg
      call msg_message
      call msg_fatal (" Model '" // char (model%name) // "'" // &
            " has no particle with this PDG code")
    end function model_get_particle_index
```

Return a pointer to the particle-data object that belongs to the specified PDG
code or name.

⟨Models: public⟩+≡
```
    public :: model_get_particle_ptr
```

⟨Models: procedures⟩+≡
```
    function model_get_particle_ptr (model, pdg) result (prt)
      type(particle_data_t), pointer :: prt
      type(model_t), intent(in), target :: model
      integer, intent(in) :: pdg
      prt => null ()
      if (pdg /= UNDEFINED) then
         prt => model%prt(model_get_particle_index (model, pdg))
      end if
    end function model_get_particle_ptr
```

Test if a particle with the given PDG code exists in the model.

⟨Models: public⟩+≡
```
    public :: model_test_particle
```

⟨Models: procedures⟩+≡
```
    function model_test_particle (model, pdg) result (exists)
      logical :: exists
      type(particle_data_t), pointer :: prt
      type(model_t), intent(in), target :: model
      integer, intent(in) :: pdg
      integer :: i
      prt => null ()
      if (pdg /= UNDEFINED) then
         do i = 1, size (model%prt)
            if (model%prt(i)%pdg == abs (pdg)) then
               prt => model%prt(i);  exit
            end if
```

```
        end do
        exists = associated(prt)
     else
        exists = .false.
     end if
  end function model_test_particle
```

Set the value, not the pointer. If the PDG code is not valid, do nothing.

⟨*Models: public*⟩+≡
```
    public :: model_set_particle_mass
    public :: model_set_particle_width
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_set_particle_mass (model, pdg, mass)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      real(default), intent(in) :: mass
      type(particle_data_t), pointer :: prt
      prt => model_get_particle_ptr (model, pdg)
      if (associated (prt))  call particle_data_set_mass (prt, mass)
    end subroutine model_set_particle_mass

    subroutine model_set_particle_width (model, pdg, width)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: pdg
      real(default), intent(in) :: width
      type(particle_data_t), pointer :: prt
      prt => model_get_particle_ptr (model, pdg)
      if (associated (prt))  call particle_data_set_width (prt, width)
    end subroutine model_set_particle_width
```

Return the PDG code that matches a particle name or belongs to a particle index.

⟨*Models: public*⟩+≡
```
    public :: model_get_particle_pdg
```

⟨*Models: interfaces*⟩+≡
```
    interface model_get_particle_pdg
       module procedure model_get_particle_pdg_name
       module procedure model_get_particle_pdg_index
    end interface model_get_particle_pdg
```

⟨*Models: procedures*⟩+≡
```
    function model_get_particle_pdg_name (model, name) result (pdg)
      integer :: pdg
      type(model_t), intent(in), target :: model
      type(string_t), intent(in) :: name
      integer :: i
      pdg = UNDEFINED
      do i = 1, size (model%prt)
         if (model%prt(i)%longname == name) then
            pdg = particle_data_get_pdg (model%prt(i));  exit
         else if (any (model%prt(i)%name == name)) then
            pdg = particle_data_get_pdg (model%prt(i));  exit
```

```fortran
      else if (any (model%prt(i)%anti == name)) then
          pdg = - particle_data_get_pdg (model%prt(i));  exit
      end if
    end do
    if (pdg == UNDEFINED) then
       write (msg_buffer, "(1x,A,1x,A)")  "Particle name =", char (name)
       call msg_message
       call msg_fatal (" Model '" // char (model%name) // "'" // &
            " has no particle with this name")
    end if
  end function model_get_particle_pdg_name

  function model_get_particle_pdg_index (model, index) result (pdg)
    type(model_t), intent(in) :: model
    integer, intent(in) :: index
    integer :: pdg
    pdg = model%prt(index)%pdg
  end function model_get_particle_pdg_index
```

Return a pointer to the variable list.

⟨*Models: public*⟩+≡
```fortran
    public :: model_get_var_list_ptr
```

⟨*Models: procedures*⟩+≡
```fortran
    function model_get_var_list_ptr (model) result (var_list)
      type(var_list_t), pointer :: var_list
      type(model_t), intent(in), target :: model
      var_list => model%var_list
    end function model_get_var_list_ptr
```

Vertex definition.

⟨*Models: interfaces*⟩+≡
```fortran
    interface model_set_vertex
       module procedure model_set_vertex_pdg
       module procedure model_set_vertex_names
    end interface
```

⟨*Models: procedures*⟩+≡
```fortran
    subroutine model_set_vertex_pdg (model, i, pdg)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      integer, dimension(:), intent(in) :: pdg
      call vertex_init (model%vtx(i), pdg, model)
    end subroutine model_set_vertex_pdg

    subroutine model_set_vertex_names (model, i, name)
      type(model_t), intent(inout), target :: model
      integer, intent(in) :: i
      type(string_t), dimension(:), intent(in) :: name
      integer, dimension(size(name)) :: pdg
      integer :: j
      do j = 1, size (name)
         pdg(j) = model_get_particle_pdg (model, name(j))
      end do
```

```
      call vertex_init (model%vtx(i), pdg, model)
    end subroutine model_set_vertex_names
```

Lookup functions

⟨*Models: public*⟩+≡
```
    public :: model_match_vertex
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_match_vertex (model, pdg1, pdg2, pdg3)
      type(model_t), intent(in) :: model
      integer, intent(in) :: pdg1, pdg2
      integer, dimension(:), allocatable, intent(out) :: pdg3
      call vertex_table_match (model%vt, pdg1, pdg2, pdg3)
    end subroutine model_match_vertex
```

⟨*Models: public*⟩+≡
```
    public :: model_check_vertex
```

⟨*Models: procedures*⟩+≡
```
    function model_check_vertex (model, pdg1, pdg2, pdg3) result (flag)
      logical :: flag
      type(model_t), intent(in) :: model
      integer, intent(in) :: pdg1, pdg2, pdg3
      flag = vertex_table_check (model%vt, pdg1, pdg2, pdg3)
    end function model_check_vertex
```

6.1.8 Reading models from file

This procedure defines the model-file syntax for the parser, returning an internal
file (ifile).

Note that arithmetic operators are defined as keywords in the expression
syntax, so we exclude them here.

⟨*Models: procedures*⟩+≡
```
    subroutine define_model_file_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ model_def = model_name_def " // &
          "parameters derived_pars external_pars particles vertices")
      call ifile_append (ifile, "SEQ model_name_def = model model_name")
      call ifile_append (ifile, "KEY model")
      call ifile_append (ifile, "QUO model_name = '""'...'""'")
      call ifile_append (ifile, "SEQ parameters = parameter_def*")
      call ifile_append (ifile, "SEQ parameter_def = parameter par_name " // &
          "'=' any_real_value")
      call ifile_append (ifile, "ALT any_real_value = " &
          // "neg_real_value | pos_real_value | real_value")
      call ifile_append (ifile, "SEQ neg_real_value = '-' real_value")
      call ifile_append (ifile, "SEQ pos_real_value = '+' real_value")
      call ifile_append (ifile, "KEY parameter")
      call ifile_append (ifile, "IDE par_name")
      ! call ifile_append (ifile, "KEY '='")          !!! Key already exists
      call ifile_append (ifile, "SEQ derived_pars = derived_def*")
      call ifile_append (ifile, "SEQ derived_def = derived par_name " // &
```

```
      "'=' expr")
    call ifile_append (ifile, "KEY derived")
    call ifile_append (ifile, "SEQ external_pars = external_def*")
    call ifile_append (ifile, "SEQ external_def = external par_name")
    call ifile_append (ifile, "KEY external")
    call ifile_append (ifile, "SEQ particles = particle_def*")
    call ifile_append (ifile, "SEQ particle_def = particle prt_longname " // &
        "prt_pdg prt_details")
    call ifile_append (ifile, "KEY particle")
    call ifile_append (ifile, "IDE prt_longname")
    call ifile_append (ifile, "INT prt_pdg")
    call ifile_append (ifile, "ALT prt_details = prt_src | prt_properties")
    call ifile_append (ifile, "SEQ prt_src = like prt_longname prt_properties")
    call ifile_append (ifile, "KEY like")
    call ifile_append (ifile, "SEQ prt_properties = prt_property*")
    call ifile_append (ifile, "ALT prt_property = " // &
        "parton | invisible | gauge | left | right | " // &
        "prt_name | prt_anti | prt_tex_name | prt_tex_anti | " // &
        "prt_spin | prt_isospin | prt_charge | " // &
        "prt_color | prt_mass | prt_width")
    call ifile_append (ifile, "KEY parton")
    call ifile_append (ifile, "KEY invisible")
    call ifile_append (ifile, "KEY gauge")
    call ifile_append (ifile, "KEY left")
    call ifile_append (ifile, "KEY right")
    call ifile_append (ifile, "SEQ prt_name = name name_def+")
    call ifile_append (ifile, "SEQ prt_anti = anti name_def+")
    call ifile_append (ifile, "SEQ prt_tex_name = tex_name name_def")
    call ifile_append (ifile, "SEQ prt_tex_anti = tex_anti name_def")
    call ifile_append (ifile, "KEY name")
    call ifile_append (ifile, "KEY anti")
    call ifile_append (ifile, "KEY tex_name")
    call ifile_append (ifile, "KEY tex_anti")
    call ifile_append (ifile, "ALT name_def = name_string | name_id")
    call ifile_append (ifile, "QUO name_string = '""'...'""'")
    call ifile_append (ifile, "IDE name_id")
    call ifile_append (ifile, "SEQ prt_spin = spin frac")
    call ifile_append (ifile, "KEY spin")
    call ifile_append (ifile, "SEQ prt_isospin = isospin frac")
    call ifile_append (ifile, "KEY isospin")
    call ifile_append (ifile, "SEQ prt_charge = charge frac")
    call ifile_append (ifile, "KEY charge")
    call ifile_append (ifile, "SEQ prt_color = color integer_literal")
    call ifile_append (ifile, "KEY color")
    call ifile_append (ifile, "SEQ prt_mass = mass par_name")
    call ifile_append (ifile, "KEY mass")
    call ifile_append (ifile, "SEQ prt_width = width par_name")
    call ifile_append (ifile, "KEY width")
    call ifile_append (ifile, "SEQ vertices = vertex_def*")
    call ifile_append (ifile, "SEQ vertex_def = vertex name_def+")
    call ifile_append (ifile, "KEY vertex")
    call define_expr_syntax (ifile, particles=.false., analysis=.false.)
  end subroutine define_model_file_syntax
```

The model-file syntax and lexer are fixed, therefore stored as module variables:

⟨*Models: variables*⟩≡
```
    type(syntax_t), target, save :: syntax_model_file
```

⟨*Models: public*⟩+≡
```
    public :: syntax_model_file_init
```

⟨*Models: procedures*⟩+≡
```
    subroutine syntax_model_file_init ()
      type(ifile_t) :: ifile
      call define_model_file_syntax (ifile)
      call syntax_init (syntax_model_file, ifile)
      call ifile_final (ifile)
    end subroutine syntax_model_file_init
```

⟨*Models: procedures*⟩+≡
```
    subroutine lexer_init_model_file (lexer)
      type(lexer_t), intent(out) :: lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"{', &
           quote_match = '"}', &
           single_chars = ":()", &
           special_class = (/ "+-*/^", "<>=  " /) , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_model_file))
    end subroutine lexer_init_model_file
```

⟨*Models: public*⟩+≡
```
    public :: syntax_model_file_final
```

⟨*Models: procedures*⟩+≡
```
    subroutine syntax_model_file_final ()
      call syntax_final (syntax_model_file)
    end subroutine syntax_model_file_final
```

⟨*Models: public*⟩+≡
```
    public :: syntax_model_file_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine syntax_model_file_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_model_file, unit)
    end subroutine syntax_model_file_write
```

⟨*Models: public*⟩+≡
```
    public :: model_read
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_read (model, filename, os_data, exist)
      type(model_t), intent(out), target :: model
      type(string_t), intent(in) :: filename
      type(os_data_t), intent(in) :: os_data
      logical, intent(out) :: exist
      type(string_t) :: file
```

660

```
type(stream_t), target :: stream
type(lexer_t) :: lexer
integer :: unit
character(32) :: model_md5sum
type(parse_node_t), pointer :: nd_model_def, nd_model_name_def
type(parse_node_t), pointer :: nd_parameters, nd_derived_pars
type(parse_node_t), pointer :: nd_external_pars
type(parse_node_t), pointer :: nd_particles, nd_vertices
type(string_t) :: model_name, lib_name
integer :: n_par, n_der, n_ext, n_prt, n_vtx
real(c_default_float), dimension(:), allocatable :: par
integer :: i
type(parse_node_t), pointer :: nd_par_def
type(parse_node_t), pointer :: nd_der_def
type(parse_node_t), pointer :: nd_ext_def
type(parse_node_t), pointer :: nd_prt
type(parse_node_t), pointer :: nd_vtx
type(pdg_array_t) :: prt_undefined
file = filename
inquire (file=char(file), exist=exist)
if ((.not. exist) .and. (.not. os_data%use_testfiles)) then
   file = os_data%whizard_modelpath_local // "/" // filename
   inquire (file = char (file), exist = exist)
end if
if (.not. exist) then
   file = os_data%whizard_modelpath // "/" // filename
   inquire (file = char (file), exist = exist)
end if
if (.not. exist) then
   call msg_fatal ("Model file '" // char (filename) // "' not found")
   return
end if
if (logging) call msg_message ("Reading model file '" // char (file) // "'")
call lexer_init_model_file (lexer)
unit = free_unit ()
open (file=char(file), unit=unit, action="read", status="old")
model_md5sum = md5sum (unit)
close (unit)
call stream_init (stream, char (file))
call lexer_assign_stream (lexer, stream)
call parse_tree_init (model%parse_tree, syntax_model_file, lexer)
call stream_final (stream)
call lexer_final (lexer)
! call parse_tree_write (parse_tree)     !!! Debugging
nd_model_def => parse_tree_get_root_ptr (model%parse_tree)
nd_model_name_def => parse_node_get_sub_ptr (nd_model_def)
model_name = parse_node_get_string &
     (parse_node_get_sub_ptr (nd_model_name_def, 2))
nd_parameters => parse_node_get_next_ptr (nd_model_name_def)
if (associated (nd_parameters)) then
   if (parse_node_get_rule_key (nd_parameters) == "parameters") then
      n_par = parse_node_get_n_sub (nd_parameters)
      nd_par_def => parse_node_get_sub_ptr (nd_parameters)
      nd_derived_pars => parse_node_get_next_ptr (nd_parameters)
```

661

```
            else
               n_par = 0
               nd_derived_pars => nd_parameters
               nd_parameters => null ()
            end if
         else
            n_par = 0
            nd_derived_pars => null ()
         end if
         if (associated (nd_derived_pars)) then
            if (parse_node_get_rule_key (nd_derived_pars) == "derived_pars") then
               n_der = parse_node_get_n_sub (nd_derived_pars)
               nd_der_def => parse_node_get_sub_ptr (nd_derived_pars)
               nd_external_pars => parse_node_get_next_ptr (nd_derived_pars)
            else
               n_der = 0
               nd_external_pars => nd_derived_pars
               nd_derived_pars => null ()
            end if
         else
            n_der = 0
            nd_external_pars => null ()
         end if
         if (associated (nd_external_pars)) then
            if (parse_node_get_rule_key (nd_external_pars) == "external_pars") then
               n_ext = parse_node_get_n_sub (nd_external_pars)
               lib_name = "external." // model_name
               nd_ext_def => parse_node_get_sub_ptr (nd_external_pars)
               nd_particles => parse_node_get_next_ptr (nd_external_pars)
            else
               n_ext = 0
               lib_name = ""
               nd_particles => nd_external_pars
               nd_external_pars => null ()
            end if
         else
            n_ext = 0
            lib_name = ""
            nd_particles => null ()
         end if
         if (associated (nd_particles)) then
            if (parse_node_get_rule_key (nd_particles) == "particles") then
               n_prt = parse_node_get_n_sub (nd_particles)
               nd_prt => parse_node_get_sub_ptr (nd_particles)
               nd_vertices => parse_node_get_next_ptr (nd_particles)
            else
               n_prt = 0
               nd_vertices => nd_particles
               nd_particles => null ()
            end if
         else
            n_prt = 0
            nd_vertices => null ()
         end if
```

```
         if (associated (nd_vertices)) then
            n_vtx = parse_node_get_n_sub (nd_vertices)
            nd_vtx => parse_node_get_sub_ptr (nd_vertices)
         else
            n_vtx = 0
         end if
         call model_init (model, model_name, lib_name, os_data, &
              n_par + n_der + n_ext, n_prt, n_vtx)
         model%md5sum = model_md5sum
         do i = 1, n_par
            call model_read_parameter (model, i, nd_par_def)
            nd_par_def => parse_node_get_next_ptr (nd_par_def)
         end do
         do i = n_par + 1, n_par + n_der
            call model_read_derived (model, i, nd_der_def)
            nd_der_def => parse_node_get_next_ptr (nd_der_def)
         end do
         do i = n_par + n_der + 1, n_par + n_der + n_ext
            call model_read_external (model, i, nd_ext_def)
            nd_ext_def => parse_node_get_next_ptr (nd_ext_def)
         end do
         if (associated (model% init_external_parameters)) then
            call model_parameters_to_c_array (model, par)
            call model% init_external_parameters (par)
            call model_parameters_from_c_array (model, par)
         end if
         prt_undefined = UNDEFINED
         call var_list_append_pdg_array &
              (model%var_list, var_str ("particle"), &
               prt_undefined, locked = .true., intrinsic=.true.)
         do i = 1, n_prt
            call model_read_particle (model, i, nd_prt)
            nd_prt => parse_node_get_next_ptr (nd_prt)
         end do
         do i = 1, n_vtx
            call model_read_vertex (model, i, nd_vtx)
            nd_vtx => parse_node_get_next_ptr (nd_vtx)
         end do
         call var_list_append_pdg_array &
              (model%var_list, var_str ("charged"), &
               particle_data_get_charged_pdg (model%prt), locked = .true., &
               intrinsic=.true.)
         call var_list_append_pdg_array &
              (model%var_list, var_str ("colored"), &
               particle_data_get_colored_pdg (model%prt), locked = .true., &
               intrinsic=.true.)
      end subroutine model_read
```

Parameters are real values (literal) with an optional unit.

⟨*Models: procedures*⟩+≡

```
      subroutine model_read_parameter (model, i, node)
        type(model_t), intent(inout), target :: model
        integer, intent(in) :: i
```

```
    type(parse_node_t), intent(in), target :: node
    type(parse_node_t), pointer :: node_name, node_val
    type(string_t) :: name
    node_name => parse_node_get_sub_ptr (node, 2)
    name = parse_node_get_string (node_name)
    node_val => parse_node_get_next_ptr (node_name, 2)
    call model_set_parameter (model, i, name, node_val, constant=.true.)
  end subroutine model_read_parameter
```

Derived parameters have any numeric expression as their definition.

⟨*Models: procedures*⟩+≡
```
  subroutine model_read_derived (model, i, node)
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in), target :: node
    type(string_t) :: name
    type(parse_node_t), pointer :: pn_expr
    name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
    pn_expr => parse_node_get_sub_ptr (node, 4)
    call model_set_parameter (model, i, name, pn_expr, constant=.false.)
  end subroutine model_read_derived
```

External parameters have no definition; they are handled by an external library.

⟨*Models: procedures*⟩+≡
```
  subroutine model_read_external (model, i, node)
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in), target :: node
    type(string_t) :: name
    name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
    call model_set_parameter_external (model, i, name)
  end subroutine model_read_external
```

⟨*Models: procedures*⟩+≡
```
  subroutine model_read_particle (model, i, node)
    type(model_t), intent(inout) :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in) :: node
    type(parse_node_t), pointer :: nd_src, nd_props, nd_prop
    type(string_t) :: longname
    integer :: pdg
    type(string_t) :: name_src
    type(string_t), dimension(:), allocatable :: name
    longname = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
    pdg = parse_node_get_integer (parse_node_get_sub_ptr (node, 3))
    call model_init_particle (model, i, longname, pdg)
    nd_src => parse_node_get_sub_ptr (node, 4)
    if (associated (nd_src)) then
       if (parse_node_get_rule_key (nd_src) == "prt_src") then
          name_src = parse_node_get_string (parse_node_get_sub_ptr (nd_src, 2))
          call model_copy_particle_data (model, i, name_src)
          nd_props => parse_node_get_sub_ptr (nd_src, 3)
       else
```

```
                  nd_props => nd_src
            end if
            nd_prop => parse_node_get_sub_ptr (nd_props)
            do while (associated (nd_prop))
               select case (char (parse_node_get_rule_key (nd_prop)))
               case ("invisible")
                  call model_set_particle_data (model, i, is_visible=.false.)
               case ("parton")
                  call model_set_particle_data (model, i, is_parton=.true.)
               case ("gauge")
                  call model_set_particle_data (model, i, is_gauge=.true.)
               case ("left")
                  call model_set_particle_data (model, i, is_left_handed=.true.)
               case ("right")
                  call model_set_particle_data (model, i, is_right_handed=.true.)
               case ("prt_name")
                  call read_names (nd_prop, name)
                  call model_set_particle_data (model, i, name=name)
               case ("prt_anti")
                  call read_names (nd_prop, name)
                  call model_set_particle_data (model, i, anti=name)
               case ("prt_tex_name")
                  call model_set_particle_data (model, i, &
                       tex_name = parse_node_get_string &
                       (parse_node_get_sub_ptr (nd_prop, 2)))
               case ("prt_tex_anti")
                  call model_set_particle_data (model, i, &
                       tex_anti = parse_node_get_string &
                       (parse_node_get_sub_ptr (nd_prop, 2)))
               case ("prt_spin")
                  call model_set_particle_data (model, i, &
                       spin_type = read_frac &
                       (parse_node_get_sub_ptr (nd_prop, 2), 2))
               case ("prt_isospin")
                  call model_set_particle_data (model, i, &
                       isospin_type = read_frac &
                       (parse_node_get_sub_ptr (nd_prop, 2), 2))
               case ("prt_charge")
                  call model_set_particle_data (model, i, &
                       charge_type = read_frac &
                       (parse_node_get_sub_ptr (nd_prop, 2), 3))
               case ("prt_color")
                  call model_set_particle_data (model, i, &
                       color_type = parse_node_get_integer &
                       (parse_node_get_sub_ptr (nd_prop, 2)))
               case ("prt_mass")
                  call model_set_particle_data (model, i, &
                       mass_src = model_get_parameter_ptr &
                       (model, parse_node_get_string &
                       (parse_node_get_sub_ptr (nd_prop, 2))))
               case ("prt_width")
                  call model_set_particle_data (model, i, &
                       width_src = model_get_parameter_ptr &
                       (model, parse_node_get_string &
```

```
                      (parse_node_get_sub_ptr (nd_prop, 2))))
            case default
               call msg_bug (" Unknown particle property '" &
                       // char (parse_node_get_rule_key (nd_prop)) // "'")
            end select
            if (allocated (name))  deallocate (name)
            nd_prop => parse_node_get_next_ptr (nd_prop)
         end do
      end if
      call model_freeze_particle_data (model, i)
    end subroutine model_read_particle
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_read_vertex (model, i, node)
      type(model_t), intent(inout) :: model
      integer, intent(in) :: i
      type(parse_node_t), intent(in) :: node
      type(string_t), dimension(:), allocatable :: name
      call read_names (node, name)
      call model_set_vertex (model, i, name)
    end subroutine model_read_vertex
```

⟨*Models: procedures*⟩+≡
```
    subroutine read_names (node, name)
      type(parse_node_t), intent(in) :: node
      type(string_t), dimension(:), allocatable, intent(inout) :: name
      type(parse_node_t), pointer :: nd_name
      integer :: n_names, i
      n_names = parse_node_get_n_sub (node) - 1
      allocate (name (n_names))
      nd_name => parse_node_get_sub_ptr (node, 2)
      do i = 1, n_names
         name(i) = parse_node_get_string (nd_name)
         nd_name => parse_node_get_next_ptr (nd_name)
      end do
    end subroutine read_names
```

⟨*Models: procedures*⟩+≡
```
    function read_frac (nd_frac, base) result (qn_type)
      integer :: qn_type
      type(parse_node_t), intent(in) :: nd_frac
      integer, intent(in) :: base
      type(parse_node_t), pointer :: nd_num, nd_den
      integer :: num, den
      nd_num => parse_node_get_sub_ptr (nd_frac)
      nd_den => parse_node_get_next_ptr (nd_num)
      select case (char (parse_node_get_rule_key (nd_num)))
      case ("integer_literal")
         num = parse_node_get_integer (nd_num)
      case ("neg_int")
         num = - parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
      case ("pos_int")
         num = parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
```

666

```
        case default
           call parse_tree_bug (nd_num, "int|neg_int|pos_int")
        end select
        if (associated (nd_den)) then
           den = parse_node_get_integer (parse_node_get_sub_ptr (nd_den, 2))
        else
           den = 1
        end if
        if (den == 1) then
           qn_type = sign (1 + abs (num) * base, num)
        else if (den == base) then
           qn_type = sign (abs (num) + 1, num)
        else
           call parse_node_write_rec (nd_frac)
           call msg_fatal (" Fractional quantum number: wrong denominator")
        end if
      end function read_frac
```

6.1.9   Model list

List of currently active models

⟨*Models: types*⟩+≡
```
    type :: model_entry_t
       type(model_t) :: model
       type(model_entry_t), pointer :: next => null ()
    end type model_entry_t
```

⟨*Models: public*⟩+≡
```
    public :: model_list_t
```

⟨*Models: types*⟩+≡
```
    type :: model_list_t
       type(model_entry_t), pointer :: first => null ()
       type(model_entry_t), pointer :: last => null ()
     contains
     ⟨Models: model list: TBP⟩
    end type model_list_t
```

Write an account of the model list.

⟨*Models: model list: TBP*⟩≡
```
    procedure :: write => model_list_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine model_list_write (object, unit, verbose)
      class(model_list_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      type(model_entry_t), pointer :: current
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      current => object%first
      if (associated (current)) then
```

```
            do while (associated (current))
               call model_write (current%model, unit, verbose)
               current => current%next
               if (associated (current))  write (u, *)
            end do
         end if
      end subroutine model_list_write
```

Add a new model with given `name` to the list, if it does not yet exist. If successful, return a pointer to the new model.

⟨*Models: model list: TBP*⟩+≡
```
      procedure :: add => model_list_add
```

⟨*Models: procedures*⟩+≡
```
      subroutine model_list_add (model_list, &
           name, os_data, n_par, n_prt, n_vtx, model)
        class(model_list_t), intent(inout) :: model_list
        type(string_t), intent(in) :: name
        type(os_data_t), intent(in) :: os_data
        integer, intent(in) :: n_par, n_prt, n_vtx
        type(model_t), pointer :: model
        type(model_entry_t), pointer :: current
        if (model_list%model_exists (name)) then
           model => null ()
        else
           allocate (current)
           if (associated (model_list%first)) then
              model_list%last%next => current
           else
              model_list%first => current
           end if
           model_list%last => current
           model => current%model
           call model_init (model, name, var_str (""), os_data, &
                n_par, n_prt, n_vtx)
        end if
      end subroutine model_list_add
```

Read a new model from file and add to the list, if it does not yet exist. Finalize the model by allocating the vertex table. Return a pointer to the new model. If unsuccessful, return the original pointer.

⟨*Models: model list: TBP*⟩+≡
```
      procedure :: read_model => model_list_read_model
```

⟨*Models: procedures*⟩+≡
```
      subroutine model_list_read_model (model_list, name, filename, os_data, model)
        class(model_list_t), intent(inout) :: model_list
        type(string_t), intent(in) :: name, filename
        type(os_data_t), intent(in) :: os_data
        type(model_t), pointer :: model
        type(model_entry_t), pointer :: current
        logical :: exist
        if (.not. model_list%model_exists (name)) then
           allocate (current)
```

```
      call model_read (current%model, filename, os_data, exist)
      if (.not. exist)  return
      if (current%model%name /= name) then
         call msg_fatal ("Model file '" // char (filename) // &
              "' contains model '" // char (current%model%name) // &
              "' instead of '" // char (name) // "'")
         call model_final (current%model);  deallocate (current)
         return
      end if
      if (associated (model_list%first)) then
         model_list%last%next => current
      else
         model_list%first => current
      end if
      model_list%last => current
      call vertex_table_init &
           (current%model%vt, current%model%prt, current%model%vtx)
      model => current%model
   else
      model => model_list%get_model_ptr (name)
   end if
end subroutine model_list_read_model
```

Check if a model exists by examining the list

⟨*Models: model list: TBP*⟩+≡
```
    procedure :: model_exists => model_list_model_exists
```

⟨*Models: procedures*⟩+≡
```
    function model_list_model_exists (model_list, name) result (exists)
      class(model_list_t), intent(in) :: model_list
      logical :: exists
      type(string_t), intent(in) :: name
      type(model_entry_t), pointer :: current
      current => model_list%first
      do while (associated (current))
         if (current%model%name == name) then
            exists = .true.
            return
         end if
         current => current%next
      end do
      exists = .false.
    end function model_list_model_exists
```

Return a pointer to a named model

⟨*Models: model list: TBP*⟩+≡
```
    procedure :: get_model_ptr => model_list_get_model_ptr
```

⟨*Models: procedures*⟩+≡
```
    function model_list_get_model_ptr (model_list, name) result (model)
      class(model_list_t), intent(in) :: model_list
      type(model_t), pointer :: model
      type(string_t), intent(in) :: name
      type(model_entry_t), pointer :: current
```

```
      current => model_list%first
      do while (associated (current))
         if (current%model%name == name) then
            model => current%model
            return
         end if
         current => current%next
      end do
      model => null ()
   end function model_list_get_model_ptr
```

Delete the list of models

⟨*Models: model list: TBP*⟩+≡
```
   procedure :: final => model_list_final
```

⟨*Models: procedures*⟩+≡
```
   subroutine model_list_final (model_list)
     class(model_list_t), intent(inout) :: model_list
     type(model_entry_t), pointer :: current
     model_list%last => null ()
     do while (associated (model_list%first))
        current => model_list%first
        model_list%first => model_list%first%next
        call model_final (current%model)
        deallocate (current)
     end do
   end subroutine model_list_final
```

6.1.10 Model instances

A model instance is a copy of a model object. The parameters are true copies.
The particle data and the variable list pointers should point to the copy, so
modifying the parameters has only a local effect. Hence, we build them up
explicitly. The vertex array is also rebuilt, it contains particle pointers. Finally,
the vertex hash table can be copied directly since it contains no pointers.

The `multiplicity` entry depends on the association of the `mass_src` entry
and therefore has to be set at the end.

The instance must carry the `target` attribute.

⟨*Models: public*⟩+≡
```
   public :: model_init_instance
```

⟨*Models: procedures*⟩+≡
```
   subroutine model_init_instance (model, orig)
     type(model_t), intent(out), target :: model
     type(model_t), intent(in) :: orig
     integer :: n_par, n_prt, n_vtx
     integer :: i, j
     type(pdg_array_t) :: prt_undefined
     type(pdg_array_t) :: aval
     n_par = size (orig%par)
     n_prt = size (orig%prt)
     n_vtx = size (orig%vtx)
```

```
call model_basic_init (model, orig%name, n_par, n_prt, n_vtx)
model%md5sum = orig%md5sum
do i = 1, n_par
   call model_copy_parameter (model, i, orig%par(i))
end do
prt_undefined = UNDEFINED
call var_list_append_pdg_array &
     (model%var_list, var_str ("particle"), &
      prt_undefined, locked = .true., intrinsic=.true.)
do i = 1, n_prt
   associate (prt => model%prt(i))
     associate (prt_src => orig%prt(i))
       call model_init_particle (model, i,  prt_src%longname, prt_src%pdg)
       call particle_data_copy (prt, prt_src)
       if (associated (prt_src%mass_src)) then
          prt%mass_src => &
               model_get_parameter_ptr (model, prt_src%mass_src%name)
          prt%mass_val => parameter_get_value_ptr (prt%mass_src)
       end if
       if (associated (prt_src%width_src)) then
          prt%width_src => &
               model_get_parameter_ptr (model, prt_src%width_src%name)
          prt%width_val => parameter_get_value_ptr (prt%width_src)
       end if
       call particle_data_set_multiplicity (prt)
     end associate
     aval = particle_data_get_pdg (prt)
     do j = 1, size (prt%name)
       call var_list_append_pdg_array &
          (model%var_list, prt%name(j), aval, locked=.true., &
          intrinsic=.true.)
     end do
     if (prt%has_antiparticle) then
        aval = - particle_data_get_pdg (prt)
        do j = 1, size (prt%anti)
          call var_list_append_pdg_array &
               (model%var_list, prt%anti(j), aval, locked=.true., &
               intrinsic=.true.)
        end do
     end if
   end associate
end do
model%init_external_parameters => orig%init_external_parameters
call model_parameters_update (model)
do i = 1, n_vtx
   call vertex_init (model%vtx(i), orig%vtx(i)%pdg, model)
end do
model%vt = orig%vt
call var_list_append_pdg_array &
     (model%var_list, var_str ("charged"), &
      particle_data_get_charged_pdg (model%prt), locked = .true., &
      intrinsic=.true.)
call var_list_append_pdg_array &
     (model%var_list, var_str ("colored"), &
```

```
                particle_data_get_colored_pdg (model%prt), locked = .true., &
                intrinsic=.true.)
        end subroutine model_init_instance
```

Turn a pointer to a model into a true copy, i.e., a new instance.

⟨*Models: public*⟩+≡
```
        public :: model_pointer_to_instance
```

⟨*Models: procedures*⟩+≡
```
        subroutine model_pointer_to_instance (model)
          type(model_t), pointer, intent(inout) :: model
          type(model_t), pointer :: model_tmp
          model_tmp => model
          allocate (model)
          call model_init_instance (model, model_tmp)
        end subroutine model_pointer_to_instance
```

6.1.11   Inclusive Processes

Automatically generate a list of possible final states for a given initial state
(currently: a single massive particle).

The current implementation is intended for simple 2- and 3-body decays
and will perform badly if used for other purposes. We generate all final states
recursively without caring about kinematics or doublers, sort each final state,
then transfer them all to an array. In this array, we mark all kinematically
forbidden channels and all doublers, to finally a packed version.

If the flag `radiative` is unset, we do look at kinematics for each branching:
a branching is omitted if either (a) all three

For temporary storage we need a list.

⟨*Models: types*⟩+≡
```
        type, extends (pdg_array_t) :: fs_entry_t
          type(fs_entry_t), pointer :: previous => null ()
          type(fs_entry_t), pointer :: next => null ()
        end type fs_entry_t
```

This is the wrapper type for the decay tree for the list of final states and the
final array.

⟨*Models: public*⟩+≡
```
        public :: fs_table_t
```

⟨*Models: types*⟩+≡
```
        type :: fs_table_t
          type(model_t), pointer :: model => null ()
          integer :: pdg_in = 0
          integer :: n_max = 0
          real(default) :: e = 0
          logical :: radiative = .false.
          type(fs_entry_t), pointer :: first => null ()
          type(fs_entry_t), pointer :: last => null ()
        contains
        ⟨Models: fs table: TBP⟩
```

```
      end type fs_table_t
```

Write the table.

⟨*Models: fs table: TBP*⟩≡
```
    procedure :: write => fs_table_write
```

⟨*Models: procedures*⟩+≡
```
    subroutine fs_table_write (object, unit)
      class(fs_table_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer, dimension(:), allocatable :: pdg
      type(fs_entry_t), pointer :: entry
      integer :: u, j, k
      u = output_unit (unit)
      k = model_get_particle_index (object%model, object%pdg_in)
      write (u, "(1x,A,1x,A)")  "Decays for particle:", &
          char (particle_data_get_name (object%model%prt(k), object%pdg_in < 0))
      entry => object%first
      do while (associated (entry))
         write (u, "(2x)", advance = "no")
         pdg = entry%pdg_array_t
         do j = 1, size (pdg)
            k = model_get_particle_index (object%model, pdg(j))
            write (u, "(1x,A)", advance = "no") &
                char (particle_data_get_name (object%model%prt(k), pdg(j) < 0))
         end do
         write (u, *)
         entry => entry%next
      end do
    end subroutine fs_table_write
```

Create the table. Construct all possible final states up to `n_max` entries, starting from a single particle with PDG code `pdg_in`.

⟨*Models: fs table: TBP*⟩+≡
```
    procedure :: make => fs_table_make
```

⟨*Models: procedures*⟩+≡
```
    subroutine fs_table_make (table, model, pdg_in, n_max, radiative)
      class(fs_table_t), intent(out) :: table
      type(model_t), intent(in), target :: model
      integer, intent(in) :: pdg_in
      integer, intent(in) :: n_max
      logical, intent(in), optional :: radiative
      type(pdg_array_t) :: pa_in
      integer :: i
      table%model => model
      table%pdg_in = pdg_in
      table%n_max = n_max
      i = model_get_particle_index (model, pdg_in)
      table%e = particle_data_get_mass (model%prt(i))
      if (present (radiative))  table%radiative = radiative
      pa_in = [pdg_in]
      call table%split (pa_in, 1)
    end subroutine fs_table_make
```

Recursive splitting. We take a PDG array and the index of an element, split this element in all possible ways, record the result, and continue splitting.

The recursion terminates whenever the split array would have a length greater than $n_{\max}$.

⟨*Models: fs table: TBP*⟩+≡
```
procedure :: split => fs_table_split
```

⟨*Models: procedures*⟩+≡
```
recursive subroutine fs_table_split (table, pa, i, record)
  class(fs_table_t), intent(inout) :: table
  type(pdg_array_t), intent(in) :: pa
  integer, intent(in) :: i
  logical, intent(in), optional :: record
  integer :: pdg1, v, l
  integer, dimension(:), allocatable :: pdg2
  if (present (record)) then
     if (record)  call table%record (sort_abs (pa))
  end if
  pdg1 = pdg_array_get (pa, i)
  l = pdg_array_get_length (pa)
  do v = 1, size (table%model%vtx)
     call vertex_get_match (table%model%vtx(v), pdg1, pdg2, table%radiative)
     if (allocated (pdg2)) then
        if (l + size (pdg2) - 1 <= table%n_max) then
           call fs_table_split (table, pdg_array_replace (pa, i, pdg2), i, &
                record = .true.)
        end if
     end if
  end do
  if (i < l) then
     call fs_table_split (table, pa, i + 1)
  end if
end subroutine fs_table_split
```

Record an entry: insert into the list. Check the ordering and insert it at the correct place, unless it is already there.

We record an array only if its mass sum is less than the total available energy.

⟨*Models: fs table: TBP*⟩+≡
```
procedure :: record => fs_table_record
```

⟨*Models: procedures*⟩+≡
```
subroutine fs_table_record (table, pa)
  class(fs_table_t), intent(inout) :: table
  type(pdg_array_t), intent(in) :: pa
  type(fs_entry_t), pointer :: current
  current => table%first
  do while (associated (current))
     if (pa == current%pdg_array_t) then
        return
     else if (mass_sum (pa, table%model) >= table%e) then
        return
     else if (pa < current%pdg_array_t) then
```

```
      call insert
      return
    end if
    current => current%next
  end do
  call insert
contains
  subroutine insert ()
    type(fs_entry_t), pointer :: entry
    allocate (entry)
    entry%pdg_array_t = pa
    if (associated (current)) then
      if (associated (current%previous)) then
        current%previous%next => entry
        entry%previous => current%previous
      else
        table%first => entry
      end if
      entry%next => current
      current%previous => entry
    else
      if (associated (table%last)) then
        table%last%next => entry
        entry%previous => table%last
      else
        table%first => entry
      end if
      table%last => entry
    end if
  end subroutine insert
end subroutine fs_table_record
```

Compute the mass sum for a PDG array.

⟨*Models: procedures*⟩+≡
```
function mass_sum (pa, model) result (m)
  type(pdg_array_t), intent(in) :: pa
  type(model_t), intent(in), target :: model
  real(default) :: m
  integer :: i, k
  m = 0
  do i = 1, pdg_array_get_length (pa)
    k = model_get_particle_index (model, pdg_array_get (pa, i))
    m = m + particle_data_get_mass (model%prt(k))
  end do
end function mass_sum
```

Return the number of generated decays.

⟨*Models: fs table: TBP*⟩+≡
```
procedure :: get_length => fs_table_get_length
```

⟨*Models: procedures*⟩+≡
```
function fs_table_get_length (fs_table) result (n)
  class(fs_table_t), intent(in) :: fs_table
```

```
      integer :: n
      type(fs_entry_t), pointer :: entry
      n = 0
      entry => fs_table%first
      do while (associated (entry))
         n = n + 1
         entry => entry%next
      end do
   end function fs_table_get_length
```

Return an allocated array of decay products (PDG codes)

⟨*Models: fs table: TBP*⟩+≡
```
   procedure :: get_pdg_out => fs_table_get_pdg_out
```

⟨*Models: procedures*⟩+≡
```
   subroutine fs_table_get_pdg_out (fs_table, i, pdg_out)
     class(fs_table_t), intent(in) :: fs_table
     integer, intent(in) :: i
     integer, dimension(:), allocatable, intent(out) :: pdg_out
     type(fs_entry_t), pointer :: entry
     integer :: n
     n = 0
     entry => fs_table%first
     do while (associated (entry))
        n = n + 1
        if (n == i) then
           allocate (pdg_out (pdg_array_get_length (entry%pdg_array_t)))
           pdg_out = entry%pdg_array_t
           exit
        end if
        entry => entry%next
     end do
   end subroutine fs_table_get_pdg_out
```

6.1.12 Unit tests

⟨*Models: public*⟩+≡
```
   public :: models_test
```

⟨*Models: tests*⟩≡
```
   subroutine models_test (u, results)
     integer, intent(in) :: u
     type(test_results_t), intent(inout) :: results
```
⟨*Models: execute tests*⟩
```
   end subroutine models_test
```

Construct a Model

Here, we construct a toy model explicitly without referring to a file.

⟨*Models: execute tests*⟩≡
```
   call test (models_1, "models_1", &
```

```
               "construct model", &
               u, results)
```

⟨*Models: tests*⟩+≡

```
    subroutine models_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(string_t) :: model_name
      type(string_t) :: x_longname
      type(string_t), dimension(2) :: parname
      type(string_t), dimension(2) :: x_name
      type(string_t), dimension(1) :: x_anti
      type(string_t) :: x_tex_name, x_tex_anti
      type(string_t) :: y_longname
      type(string_t), dimension(2) :: y_name
      type(string_t) :: y_tex_name

      write (u, "(A)")  "* Test output: models_1"
      write (u, "(A)")  "*   Purpose: create a model"
      write (u, *)

      model_name = "Test model"
      call model_list%add (model_name, os_data, 2, 2, 3, model)
      parname(1) = "mx"
      parname(2) = "coup"
      call model_set_parameter (model, 1, parname(1), 10._default)
      call model_set_parameter (model, 2, parname(2), 1.3_default)
      x_longname = "X_LEPTON"
      x_name(1) = "X"
      x_name(2) = "x"
      x_anti(1) = "Xbar"
      x_tex_name = "X^+"
      x_tex_anti = "X^-"
      call model_init_particle (model, 1, x_longname, 99)
      call model_set_particle_data (model, 1, &
           .true., .false., .false., .false., .false., &
           x_name, x_anti, x_tex_name, x_tex_anti, &
           SPINOR, -3, 2, 1, model_get_parameter_ptr (model, parname(1)))
      y_longname = "Y_COLORON"
      y_name(1) = "Y"
      y_name(2) = "yc"
      y_tex_name = "Y^0"
      call model_init_particle (model, 2, y_longname, 97)
      call model_set_particle_data (model, 2,  &
           .false., .false., .true., .false., .false., &
           name=y_name, tex_name=y_tex_name, &
           spin_type=SCALAR, isospin_type=2, charge_type=1, color_type=8)
      call model_set_vertex (model, 1, (/ 99, 99, 99 /))
      call model_set_vertex (model, 2, (/ 99, 99, 99, 99 /))
      call model_set_vertex (model, 3, (/ 99, 97, 99 /))
      call model_list%write (u)

      call model_list%final ()
```

677

```
      write (u, *)
      write (u, "(A)")  "* Test output end: models_1"

   end subroutine models_1
```

Read a Model

Read a predefined model from file.

⟨Models: execute tests⟩+≡
```
      call test (models_2, "models_2", &
           "read model", &
           u, results)
```

⟨Models: tests⟩+≡
```
   subroutine models_2 (u)
     integer, intent(in) :: u
     type(os_data_t) :: os_data
     type(model_list_t) :: model_list
     type(model_t), pointer :: model

     write (u, "(A)")  "* Test output: models_2"
     write (u, "(A)")  "*   Purpose: read a model from file"
     write (u, *)

     call syntax_model_file_init ()
     call os_data_init (os_data)

     call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
          os_data, model)
     call model_list%write (u)

     write (u, *)
     write (u, "(A)")  "* Variable list"
     write (u, *)

     call var_list_write (model%var_list, u)

     write (u, *)
     write (u, "(A)")  "* Cleanup"

     call model_list%final ()
     call syntax_model_file_final ()

     write (u, *)
     write (u, "(A)")  "* Test output end: models_2"

   end subroutine models_2
```

Model Instance

Read a predefined model from file and create an instance.

⟨*Models: execute tests*⟩+≡
```
    call test (models_3, "models_3", &
         "model instance", &
         u, results)
```
⟨*Models: tests*⟩+≡
```
    subroutine models_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(model_t), pointer :: instance

      write (u, "(A)")  "* Test output: models_3"
      write (u, "(A)")  "*   Purpose: create a model instance"
      write (u, *)

      call syntax_model_file_init ()
      call os_data_init (os_data)

      call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
           os_data, model)
      allocate (instance)
      call model_init_instance (instance, model)

      call model_write (instance, u)

      write (u, *)
      write (u, "(A)")  "* Variable list"
      write (u, *)

      call var_list_write (instance%var_list, u)

      write (u, *)
      write (u, "(A)")  "* Cleanup"

      call model_final (instance)
      deallocate (instance)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, *)
      write (u, "(A)")  "* Test output end: models_3"

    end subroutine models_3
```

Unstable and Polarized Particles

Read a predefined model from file and define decays and polarization.

⟨*Models: execute tests*⟩+≡
```
    call test (models_4, "models_4", &
         "handle decays and polarization", &
```

```
          u, results)
```

⟨*Models: tests*⟩+≡

```
    subroutine models_4 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model, model_instance
      character(32) :: md5sum

      write (u, "(A)")  "* Test output: models_4"
      write (u, "(A)")  "*   Purpose: set and unset decays and polarization"
      write (u, *)

      call syntax_model_file_init ()
      call os_data_init (os_data)

      write (u, "(A)")  "* Read model from file"

      call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
           os_data, model)

      md5sum = model_get_parameters_md5sum (model)
      write (u, *)
      write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

      write (u, *)
      write (u, "(A)")  "* Set particle decays and polarization"
      write (u, *)

      call model_set_unstable (model, 25, [var_str ("dec1"), var_str ("dec2")])
      call model_set_polarized (model, 6)
      call model_set_unstable (model, -6, [var_str ("fdec")])

      call model_write (model, u)

      md5sum = model_get_parameters_md5sum (model)
      write (u, *)
      write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

      write (u, *)
      write (u, "(A)")  "* Create a model instance"

      allocate (model_instance)
      call model_init_instance (model_instance, model)

      write (u, *)
      write (u, "(A)")  "* Revert particle decays and polarization"
      write (u, *)

      call model_set_stable (model, 25)
      call model_set_unpolarized (model, 6)
      call model_set_stable (model, -6)

      call model_write (model, u)
```

```
      md5sum = model_get_parameters_md5sum (model)
      write (u, *)
      write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

      write (u, *)
      write (u, "(A)")  "* Show the model instance"
      write (u, *)

      call model_write (model_instance, u)

      md5sum = model_get_parameters_md5sum (model_instance)
      write (u, *)
      write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

      write (u, *)
      write (u, "(A)")  "* Cleanup"

      call model_final (model_instance)
      deallocate (model_instance)
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, *)
      write (u, "(A)")  "* Test output end: models_4"

    end subroutine models_4
```

Generate Decay Table

Determine all kinematically allowed decay channels for a Higgs boson, using
default parameter values.

⟨*Models: execute tests*⟩+≡
```
    call test (models_5, "models_5", &
         "generate decay table", &
         u, results)
```

⟨*Models: tests*⟩+≡
```
    subroutine models_5 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(fs_table_t) :: fs_table

      write (u, "(A)")  "* Test output: models_5"
      write (u, "(A)")  "*   Purpose: determine Higgs decay table"
      write (u, *)

      call syntax_model_file_init ()
      call os_data_init (os_data)

      write (u, "(A)")  "* Read Standard Model"
```

```
    call model_list%read_model (var_str ("SM"), var_str ("SM.mdl"), &
         os_data, model)

    write (u, *)
    write (u, "(A)")  "* Higgs decays n = 2"
    write (u, *)

    call fs_table%make (model, 25, 2)
    call fs_table%write (u)

    write (u, *)
    write (u, "(A)")  "* Higgs decays n = 3 (w/o radiative)"
    write (u, *)

    call fs_table%make (model, 25, 3, radiative = .false.)
    call fs_table%write (u)

    write (u, *)
    write (u, "(A)")  "* Higgs decays n = 3 (w/ radiative)"
    write (u, *)

    call fs_table%make (model, 25, 3, radiative = .true.)
    call fs_table%write (u)

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_5"

  end subroutine models_5
```

Chapter 7

Quantum Numbers

We introduce separate types and modules for particle quantum numbers and use them for defining independent particles and entangled states.

helicities Types and methods for spin density matrices.

colors Dealing with colored particles, using the color-flow representation.

flavors PDG codes and particle properties, depends on the model.

quantum_states Quantum numbers and density matrices for entangled particle systems.

7.1 Helicities

This module defines types and tools for dealing with helicity information.

⟨`helicities.f90`⟩≡
  ⟨*File header*⟩

  ```
  module helicities
  ```

  ⟨*Use file utils*⟩

  ⟨*Standard module head*⟩

  ⟨*Helicities: public*⟩

  ⟨*Helicities: types*⟩

  ⟨*Helicities: interfaces*⟩

  ```
  contains
  ```

  ⟨*Helicities: procedures*⟩

  ```
  end module helicities
  ```

7.1.1 Helicity types

Helicities may be defined or undefined, corresponding to a polarized or unpolarized state. Each helicity is actually a pair of helicities, corresponding to an entry in the spin density matrix. Obviously, diagonal entries are distinguished. In addition, we have a ghost flag that would apply to FP ghosts in particular.

⟨*Helicities: public*⟩≡
  ```
  public :: helicity_t
  ```

⟨*Helicities: types*⟩≡
  ```
  type :: helicity_t
    private
    logical :: defined = .false.
    integer :: h1, h2
    logical :: ghost = .false.
  end type helicity_t
  ```

Initializers:

⟨*Helicities: public*⟩+≡
  ```
  public :: helicity_init
  ```

⟨*Helicities: interfaces*⟩≡
  ```
  interface helicity_init
    module procedure helicity_init0, helicity_init0g
    module procedure helicity_init1, helicity_init1g
    module procedure helicity_init2, helicity_init2g
  end interface
  ```

⟨*Helicities: procedures*⟩≡
```
    elemental subroutine helicity_init0 (hel)
      type(helicity_t), intent(out) :: hel
    end subroutine helicity_init0

    elemental subroutine helicity_init0g (hel, ghost)
      type(helicity_t), intent(out) :: hel
      logical, intent(in) :: ghost
      hel%ghost = ghost
    end subroutine helicity_init0g

    elemental subroutine helicity_init1 (hel, h)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h
      hel%defined = .true.
      hel%h1 = h
      hel%h2 = h
    end subroutine helicity_init1

    elemental subroutine helicity_init1g (hel, h, ghost)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h
      logical, intent(in) :: ghost
      call helicity_init1 (hel, h)
      hel%ghost = ghost
    end subroutine helicity_init1g

    elemental subroutine helicity_init2 (hel, h2, h1)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h1, h2
      hel%defined = .true.
      hel%h2 = h2
      hel%h1 = h1
    end subroutine helicity_init2

    elemental subroutine helicity_init2g (hel, h2, h1, ghost)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: h1, h2
      logical, intent(in) :: ghost
      call helicity_init2 (hel, h2, h1)
      hel%ghost = ghost
    end subroutine helicity_init2g
```

Set the ghost property separately:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_set_ghost
```

⟨*Helicities: procedures*⟩+≡
```
    elemental subroutine helicity_set_ghost (hel, ghost)
      type(helicity_t), intent(inout) :: hel
      logical, intent(in) :: ghost
      hel%ghost = ghost
    end subroutine helicity_set_ghost
```

Undefine:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_undefine
```

⟨*Helicities: procedures*⟩+≡
```
    elemental subroutine helicity_undefine (hel)
      type(helicity_t), intent(inout) :: hel
      hel%defined = .false.
      hel%ghost = .false.
    end subroutine helicity_undefine
```

Diagonalize by removing the second entry (use with care!)

⟨*Helicities: public*⟩+≡
```
    public :: helicity_diagonalize
```

⟨*Helicities: procedures*⟩+≡
```
    elemental subroutine helicity_diagonalize (hel)
      type(helicity_t), intent(inout) :: hel
      hel%h2 = hel%h1
    end subroutine helicity_diagonalize
```

Output (no linebreak). No output if undefined.

⟨*Helicities: public*⟩+≡
```
    public :: helicity_write
```

⟨*Helicities: procedures*⟩+≡
```
    subroutine helicity_write (hel, unit)
      type(helicity_t), intent(in) :: hel
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (hel%defined) then
         if (hel%ghost) then
            write (u, "(A)", advance="no")  "h*("
         else
            write (u, "(A)", advance="no")  "h("
         end if
         write (u, "(I0)", advance="no")  hel%h1
         if (hel%h1 /= hel%h2) then
            write (u, "(A)", advance="no") "|"
            write (u, "(I0)", advance="no")  hel%h2
         end if
         write (u, "(A)", advance="no")  ")"
      else if (hel%ghost) then
         write (u, "(A)", advance="no")  "h*"
      end if
    end subroutine helicity_write
```

Binary I/O. Write contents only if defined.

⟨*Helicities: public*⟩+≡
```
    public :: helicity_write_raw
    public :: helicity_read_raw
```

⟨*Helicities: procedures*⟩+≡
```
    subroutine helicity_write_raw (hel, u)
      type(helicity_t), intent(in) :: hel
      integer, intent(in) :: u
      write (u) hel%defined
      if (hel%defined) then
         write (u) hel%h1, hel%h2
         write (u) hel%ghost
      end if
    end subroutine helicity_write_raw

    subroutine helicity_read_raw (hel, u, iostat)
      type(helicity_t), intent(out) :: hel
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) hel%defined
      if (hel%defined) then
         read (u, iostat=iostat) hel%h1, hel%h2
         read (u, iostat=iostat) hel%ghost
      end if
    end subroutine helicity_read_raw
```

7.1.2 Predicates

Check if the helicity is defined:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_is_defined
```
⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_is_defined (hel) result (defined)
      logical :: defined
      type(helicity_t), intent(in) :: hel
      defined = hel%defined
    end function helicity_is_defined
```

Return true if the two helicities are equal or the particle is unpolarized:

⟨*Helicities: public*⟩+≡
```
    public :: helicity_is_diagonal
```
⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_is_diagonal (hel) result (diagonal)
      logical :: diagonal
      type(helicity_t), intent(in) :: hel
      if (hel%defined) then
         diagonal = hel%h1 == hel%h2
      else
         diagonal = .true.
      end if
    end function helicity_is_diagonal
```

Return the ghost flag

⟨*Helicities: public*⟩+≡
```
    public :: helicity_is_ghost
```

⟨*Helicities: procedures*⟩+≡
```
elemental function helicity_is_ghost (hel) result (ghost)
  logical :: ghost
  type(helicity_t), intent(in) :: hel
  ghost = hel%ghost
end function helicity_is_ghost
```

7.1.3 Accessing contents

This returns a two-element array and thus cannot be elemental. The result is unpredictable if the helicity is undefined.

⟨*Helicities: public*⟩+≡
```
public :: helicity_get
```

⟨*Helicities: procedures*⟩+≡
```
pure function helicity_get (hel) result (h)
  integer, dimension(2) :: h
  type(helicity_t), intent(in) :: hel
  h(1) = hel%h2
  h(2) = hel%h1
end function helicity_get
```

7.1.4 Comparisons

When comparing helicities, if either one is undefined, they are considered to match. In other words, an unpolarized particle matches any polarization. In the `dmatch` variant, it matches only diagonal helicity.

The ghost flag is ignored when matching, but matters when testing for equality.

⟨*Helicities: public*⟩+≡
```
public :: operator(.match.)
public :: operator(.dmatch.)
public :: operator(==)
public :: operator(/=)
```

⟨*Helicities: interfaces*⟩+≡
```
interface operator(.match.)
  module procedure helicity_match
end interface
interface operator(.dmatch.)
  module procedure helicity_match_diagonal
end interface
interface operator(==)
  module procedure helicity_eq
end interface
interface operator(/=)
  module procedure helicity_neq
end interface
```

⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_match (hel1, hel2) result (eq)
      logical :: eq
      type(helicity_t), intent(in) :: hel1, hel2
      if (hel1%defined .and. hel2%defined) then
         eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
      else
         eq = .true.
      end if
    end function helicity_match

    elemental function helicity_match_diagonal (hel1, hel2) result (eq)
      logical :: eq
      type(helicity_t), intent(in) :: hel1, hel2
      if (hel1%defined .and. hel2%defined) then
         eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
      else if (hel1%defined) then
         eq = hel1%h1 == hel1%h2
      else if (hel2%defined) then
         eq = hel2%h1 == hel2%h2
      else
         eq = .true.
      end if
    end function helicity_match_diagonal
```

⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_eq (hel1, hel2) result (eq)
      logical :: eq
      type(helicity_t), intent(in) :: hel1, hel2
      if (hel1%defined .and. hel2%defined) then
         eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2) &
              .and. (hel1%ghost .eqv. hel2%ghost)
      else if (.not. hel1%defined .and. .not. hel2%defined) then
         eq = hel1%ghost .eqv. hel2%ghost
      else
         eq = .false.
      end if
    end function helicity_eq
```

⟨*Helicities: procedures*⟩+≡
```
    elemental function helicity_neq (hel1, hel2) result (neq)
      logical :: neq
      type(helicity_t), intent(in) :: hel1, hel2
      if (hel1%defined .and. hel2%defined) then
         neq = (hel1%h1 /= hel2%h1) .or. (hel1%h2 /= hel2%h2) &
              .or. (hel1%ghost .neqv. hel2%ghost)
      else if (.not. hel1%defined .and. .not. hel2%defined) then
         neq = hel1%ghost .neqv. hel2%ghost
      else
         neq = .true.
      end if
    end function helicity_neq
```

689

7.1.5 Tools

Merge two helicity objects by taking the first entry from the first and the second entry from the second argument. Makes sense only if the input helicities were defined and diagonal. The handling of ghost flags is not well-defined; one should verify beforehand that they match.

⟨*Helicities: public*⟩+≡
```
public :: operator(.merge.)
```

⟨*Helicities: interfaces*⟩+≡
```
interface operator(.merge.)
   module procedure merge_helicities
end interface
```

⟨*Helicities: procedures*⟩+≡
```
elemental function merge_helicities (hel1, hel2) result (hel)
  type(helicity_t) :: hel
  type(helicity_t), intent(in) :: hel1, hel2
  if (helicity_is_defined (hel1) .and. helicity_is_defined (hel2)) then
     call helicity_init2g (hel, hel2%h1, hel1%h1, hel1%ghost)
  else if (helicity_is_defined (hel1)) then
     hel = hel1
  else if (helicity_is_defined (hel2)) then
     hel = hel2
  end if
end function merge_helicities
```

7.2 Colors

This module defines a type and tools for dealing with color information.

Each particle can have zero or more (in practice, usually not more than two) color indices. Color indices are positive; flow direction can be determined from the particle nature.

While parton shower matrix elements are diagonal in color, some special applications (e.g., subtractions for NLO matrix elements) require non-diagonal color matrices.

⟨colors.f90⟩≡
  ⟨*File header*⟩

  `module colors`

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
    `use diagnostics !NODEP!`
    `use unit_tests`

  ⟨*Standard module head*⟩

  ⟨*Colors: public*⟩

  ⟨*Colors: types*⟩

  ⟨*Colors: interfaces*⟩

  `contains`

  ⟨*Colors: procedures*⟩

  ⟨*Colors: tests*⟩

  `end module colors`

7.2.1 The color type

A particle may have an arbitrary number of color indices (in practice, from zero to two, but more are possible). This object acts as a container.

The fact that color comes as an array prohibits elemental procedures in some places. (May add interfaces and multi versions where necessary.)

The color may be undefined; this corresponds to unallocated arrays.

NOTE: Due to a compiler bug in nagfor 5.2, we do not use allocatable but fixed-size arrays with dimension 2. Only nonzero entries count. This may be more efficient anyway, but gives up some flexibility. However, the squaring algorithm currently works only for singlets, (anti)triplets and octets anyway, so two components are enough.

⟨*Colors: public*⟩≡
    `public :: color_t`

⟨*Colors: types*⟩≡
    `type :: color_t`

691

```
      private
      integer, dimension(2) :: c1 = 0, c2 = 0
      logical :: ghost = .false.
    end type color_t
```

Initializers:

⟨*Colors: public*⟩+≡
```
    public :: color_init
```

⟨*Colors: interfaces*⟩≡
```
    interface color_init
       module procedure color_init_undefined, color_init_undefined_ghost
       module procedure color_init_array, color_init_array_ghost
       module procedure color_init_arrays, color_init_arrays_ghost
    end interface
```

Undefined color: array remains unallocated

⟨*Colors: procedures*⟩≡
```
    pure subroutine color_init_undefined (col)
      type(color_t), intent(out) :: col
    end subroutine color_init_undefined

    pure subroutine color_init_undefined_ghost (col, ghost)
      type(color_t), intent(out) :: col
      logical, intent(in) :: ghost
      col%ghost = ghost
    end subroutine color_init_undefined_ghost
```

This defines color from an arbitrary length color array, suitable for any representation. We may have two color arrays (non-diagonal matrix elements). This cannot be elemental. The third version assigns an array of colors, using a two-dimensional array as input.

⟨*Colors: procedures*⟩+≡
```
    pure subroutine color_init_array (col, c1)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      col%c1 = pack (c1, c1 /= 0, col%c1)
      col%c2 = col%c1
    end subroutine color_init_array

    pure subroutine color_init_array_ghost (col, c1, ghost)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      logical, intent(in) :: ghost
      call color_init_array (col, c1)
      col%ghost = ghost
    end subroutine color_init_array_ghost

    pure subroutine color_init_arrays (col, c1, c2)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1, c2
      if (size (c1) == size (c2)) then
```

```
            col%c1 = pack (c1, c1 /= 0, col%c1)
            col%c2 = pack (c2, c2 /= 0, col%c2)
         else if (size (c1) /= 0) then
            col%c1 = pack (c1, c1 /= 0, col%c1)
            col%c2 = col%c1
         else if (size (c2) /= 0) then
            col%c1 = pack (c2, c2 /= 0, col%c2)
            col%c2 = col%c1
         end if
      end subroutine color_init_arrays

      pure subroutine color_init_arrays_ghost (col, c1, c2, ghost)
        type(color_t), intent(out) :: col
        integer, dimension(:), intent(in) :: c1, c2
        logical, intent(in) :: ghost
        call color_init_arrays (col, c1, c2)
        col%ghost = ghost
      end subroutine color_init_arrays_ghost
```

This version is restricted to singlets, triplets, antitriplets, and octets: The input contains the color and anticolor index, each of the may be zero.

⟨*Colors: public*⟩+≡
```
      public :: color_init_col_acl
```

⟨*Colors: procedures*⟩+≡
```
      elemental subroutine color_init_col_acl (col, col_in, acl_in)
        type(color_t), intent(out) :: col
        integer, intent(in) :: col_in, acl_in
        integer, dimension(0) :: null_array
        select case (col_in)
        case (0)
           select case (acl_in)
           case (0)
              call color_init_array (col, null_array)
           case default
              call color_init_array (col, (/ -acl_in /))
           end select
        case default
           select case (acl_in)
           case (0)
              call color_init_array (col, (/ col_in /))
           case default
              call color_init_array (col, (/ col_in, -acl_in /))
           end select
        end select
      end subroutine color_init_col_acl
```

This version is used for the external interface. We convert a fixed-size array of colors (for each particle) to the internal form by packing only the nonzero entries.

⟨*Colors: public*⟩+≡
```
      public :: color_init_from_array
```

⟨*Colors: interfaces*⟩+≡
```
    interface color_init_from_array
       module procedure color_init_from_array1, color_init_from_array1g
       module procedure color_init_from_array2, color_init_from_array2g
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    pure subroutine color_init_from_array1 (col, c1)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      logical, dimension(size(c1)) :: mask
      mask = c1 /= 0
      col%c1 = pack (c1, mask, col%c1)
      col%c2 = col%c1
    end subroutine color_init_from_array1

    pure subroutine color_init_from_array1g (col, c1, ghost)
      type(color_t), intent(out) :: col
      integer, dimension(:), intent(in) :: c1
      logical, intent(in) :: ghost
      call color_init_from_array1 (col, c1)
      col%ghost = ghost
    end subroutine color_init_from_array1g

    pure subroutine color_init_from_array2 (col, c1)
      integer, dimension(:,:), intent(in) :: c1
      type(color_t), dimension(size(c1,2)), intent(out) :: col
      integer :: i
      do i = 1, size (c1,2)
         call color_init_from_array1 (col(i), c1(:,i))
      end do
    end subroutine color_init_from_array2

    pure subroutine color_init_from_array2g (col, c1, ghost)
      integer, dimension(:,:), intent(in) :: c1
      type(color_t), dimension(size(c1,2)), intent(out) :: col
      logical, intent(in), dimension(:) :: ghost
      call color_init_from_array2 (col, c1)
      col%ghost = ghost
    end subroutine color_init_from_array2g
```

Set the ghost property

⟨*Colors: public*⟩+≡
```
    public :: color_set_ghost
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_set_ghost (col, ghost)
      type(color_t), intent(inout) :: col
      logical, intent(in) :: ghost
      col%ghost = ghost
    end subroutine color_set_ghost
```

694

Undefine the color state:

⟨*Colors: public*⟩+≡
```
    public :: color_undefine
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_undefine (col, undefine_ghost)
      type(color_t), intent(inout) :: col
      logical, intent(in), optional :: undefine_ghost
      col%c1 = 0
      col%c2 = 0
      if (present (undefine_ghost)) then
         if (undefine_ghost)  col%ghost = .false.
      else
         col%ghost = .false.
      end if
    end subroutine color_undefine
```

Output. As dense as possible, no linebreak. If color is undefined, no output.

⟨*Colors: public*⟩+≡
```
    public :: color_write
```

⟨*Colors: interfaces*⟩+≡
```
    interface color_write
       module procedure color_write_single
       module procedure color_write_array
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_write_single (col, unit)
      type(color_t), intent(in) :: col
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (color_is_defined (col)) then
         write (u, "(A)", advance="no")  "c("
         if (col%c1(1) /= 0)  write (u, "(I0)", advance="no")  col%c1(1)
         if (any (col%c1 /= 0))  write (u, "(1x)", advance="no")
         if (col%c1(2) /= 0)  write (u, "(I0)", advance="no")  col%c1(2)
         if (.not. color_is_diagonal (col)) then
            write (u, "(A)", advance="no")  "|"
            if (col%c2(1) /= 0)  write (u, "(I0)", advance="no")  col%c2(1)
            if (any (col%c2 /= 0))  write (u, "(1x)", advance="no")
            if (col%c2(2) /= 0)  write (u, "(I0)", advance="no")  col%c2(2)
         end if
         write (u, "(A)", advance="no")  ")"
      else if (col%ghost) then
         write (u, "(A)", advance="no")  "c*"
      end if
    end subroutine color_write_single

    subroutine color_write_array (col, unit)
      type(color_t), dimension(:), intent(in) :: col
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i
```

```
    u = output_unit (unit);  if (u < 0)  return
    write (u, "(A)", advance="no") "["
    do i = 1, size (col)
       if (i > 1)  write (u, "(1x)", advance="no")
       call color_write_single (col(i), u)
    end do
    write (u, "(A)", advance="no") "]"
end subroutine color_write_array
```

Binary I/O. For allocatable colors, this would have to be modified.

⟨*Colors: public*⟩+≡
```
    public :: color_write_raw
    public :: color_read_raw
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_write_raw (col, u)
      type(color_t), intent(in) :: col
      integer, intent(in) :: u
      logical :: defined
      defined = color_is_defined (col) .or. color_is_ghost (col)
      write (u) defined
      if (defined) then
         write (u) col%c1, col%c2
         write (u) col%ghost
      end if
    end subroutine color_write_raw

    subroutine color_read_raw (col, u, iostat)
      type(color_t), intent(out) :: col
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      logical :: defined
      read (u, iostat=iostat) defined
      if (defined) then
         read (u, iostat=iostat) col%c1, col%c2
         read (u, iostat=iostat) col%ghost
      end if
    end subroutine color_read_raw
```

7.2.2 Predicates

Return the definition status

⟨*Colors: public*⟩+≡
```
    public :: color_is_defined
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_is_defined (col) result (defined)
      logical :: defined
      type(color_t), intent(in) :: col
      defined = any (col%c1 /= 0)
    end function color_is_defined
```

Diagonal color objects have only one array allocated:

⟨*Colors: public*⟩+≡
```
    public :: color_is_diagonal
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_is_diagonal (col) result (diagonal)
      logical :: diagonal
      type(color_t), intent(in) :: col
      if (color_is_defined (col)) then
          diagonal = all (col%c1 == col%c2)
      else
          diagonal = .true.
      end if
    end function color_is_diagonal
```

Return the ghost flag

⟨*Colors: public*⟩+≡
```
    public :: color_is_ghost
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_is_ghost (col) result (ghost)
      logical :: ghost
      type(color_t), intent(in) :: col
      ghost = col%ghost
    end function color_is_ghost
```

The ghost parity: true if the color-ghost flag is set.

⟨*Colors: interfaces*⟩+≡
```
    interface color_ghost_parity
       module procedure color_ghost_parity0
       module procedure color_ghost_parity1
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    pure function color_ghost_parity0 (col) result (parity)
      type(color_t), intent(in) :: col
      logical :: parity
      parity = color_is_ghost (col)
    end function color_ghost_parity0

    pure function color_ghost_parity1 (col) result (parity)
      type(color_t), dimension(:), intent(in) :: col
      logical :: parity
      logical, dimension(size(col)) :: p
      integer :: i
      forall (i = 1:size(col))
         p(i) = color_ghost_parity0 (col(i))
      end forall
      parity = mod (count (p), 2) == 1
    end function color_ghost_parity1
```

7.2.3 Accessing contents

Return the number of color indices. We assume that it is identical for both arrays.

⟨*Colors: procedures*⟩+≡

```fortran
elemental function color_number (col) result (n)
  integer :: n
  type(color_t), intent(in) :: col
  n = count (col%c1 /= 0)
end function color_number
```

Return the (first) color/anticolor entry (assuming that color is diagonal). The result is a positive color index.

⟨*Colors: public*⟩+≡

```fortran
public :: color_get_col
public :: color_get_acl
```

⟨*Colors: procedures*⟩+≡

```fortran
elemental function color_get_col (col) result (c)
  integer :: c
  type(color_t), intent(in) :: col
  integer :: i
  do i = 1, size (col%c1)
     if (col%c1(i) > 0) then
        c = col%c1(i)
        return
     end if
  end do
  c = 0
end function color_get_col

elemental function color_get_acl (col) result (c)
  integer :: c
  type(color_t), intent(in) :: col
  integer :: i
  do i = 1, size (col%c1)
     if (col%c1(i) < 0) then
        c = - col%c1(i)
        return
     end if
  end do
  c = 0
end function color_get_acl
```

Return the color index with highest absolute value

⟨*Colors: public*⟩+≡

```fortran
public :: color_get_max_value
```

⟨*Colors: interfaces*⟩+≡

```fortran
interface color_get_max_value
   module procedure color_get_max_value0
   module procedure color_get_max_value1
   module procedure color_get_max_value2
end interface
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_get_max_value0 (col) result (cmax)
      integer :: cmax
      type(color_t), intent(in) :: col
      cmax = maxval (abs (col%c1))
    end function color_get_max_value0

    pure function color_get_max_value1 (col) result (cmax)
      integer :: cmax
      type(color_t), dimension(:), intent(in) :: col
      cmax = maxval (color_get_max_value0 (col))
    end function color_get_max_value1

    function color_get_max_value2 (col) result (cmax)
      integer :: cmax
      type(color_t), dimension(:,:), intent(in) :: col
      integer, dimension(size(col, 2)) :: cm
      integer :: i
      forall (i = 1:size(col, 2))
         cm(i) = color_get_max_value1 (col(:,i))
      end forall
      cmax = maxval (cm)
    end function color_get_max_value2
```

7.2.4 Comparisons

Similar to helicities, colors match if they are equal, or if either one is undefined.

⟨*Colors: public*⟩+≡
```
    public :: operator(.match.)
    public :: operator(==)
    public :: operator(/=)
```

⟨*Colors: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure color_match
    end interface
    interface operator(==)
       module procedure color_eq
    end interface
    interface operator(/=)
       module procedure color_neq
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_match (col1, col2) result (eq)
      logical :: eq
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
            eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2)
      else
         eq = .true.
      end if
```

```
    end function color_match

    elemental function color_eq (col1, col2) result (eq)
      logical :: eq
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
            eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2) &
                  .and. (col1%ghost .eqv. col2%ghost)
      else if (.not. color_is_defined (col1) &
         .and. .not. color_is_defined (col2)) then
         eq = col1%ghost .eqv. col2%ghost
      else
         eq = .false.
      end if
    end function color_eq
```

⟨*Colors: procedures*⟩+≡
```
    elemental function color_neq (col1, col2) result (neq)
      logical :: neq
      type(color_t), intent(in) :: col1, col2
      if (color_is_defined (col1) .and. color_is_defined (col2)) then
!        if (size (col1%c1) == size (col2%c1)) then
           neq = any (col1%c1 /= col2%c1) .or. any (col1%c2 /= col2%c2) &
                 .or. (col1%ghost .neqv. col2%ghost)
!        else
!           neq = .true.
!        end if
      else if (.not. color_is_defined (col1) &
         .and. .not. color_is_defined (col2)) then
         neq = col1%ghost .neqv. col2%ghost
      else
         neq = .true.
      end if
    end function color_neq
```

7.2.5 Tools

Shift color indices by a common offset.

⟨*Colors: public*⟩+≡
```
    public :: color_add_offset
```

⟨*Colors: procedures*⟩+≡
```
    elemental subroutine color_add_offset (col, offset)
      type(color_t), intent(inout) :: col
      integer, intent(in) :: offset
      where (col%c1 /= 0)  col%c1 = col%c1 + sign (offset, col%c1)
      where (col%c2 /= 0)  col%c2 = col%c2 + sign (offset, col%c2)
    end subroutine color_add_offset
```

Reassign color indices for an array of colored particle in canonical order. The
allocated size of the color map is such that two colors per particle can be acco-

modated.

⟨Colors: public⟩+≡

```
    public :: color_canonicalize
```

⟨Colors: procedures⟩+≡

```
    subroutine color_canonicalize (col)
      type(color_t), dimension(:), intent(inout) :: col
      integer, dimension(2*size(col)) :: map
      integer :: n_col, i, j, k
      n_col = 0
      do i = 1, size (col)
         do j = 1, size (col(i)%c1)
            if (col(i)%c1(j) /= 0) then
               k = find (abs (col(i)%c1(j)), map(:n_col))
               if (k == 0) then
                  n_col = n_col + 1
                  map(n_col) = abs (col(i)%c1(j))
                  k = n_col
               end if
               col(i)%c1(j) = sign (k, col(i)%c1(j))
            end if
            if (col(i)%c2(j) /= 0) then
               k = find (abs (col(i)%c2(j)), map(:n_col))
               if (k == 0) then
                  n_col = n_col + 1
                  map(n_col) = abs (col(i)%c2(j))
                  k = n_col
               end if
               col(i)%c2(j) = sign (k, col(i)%c2(j))
            end if
         end do
      end do
    contains
      function find (c, array) result (k)
        integer :: k
        integer, intent(in) :: c
        integer, dimension(:), intent(in) :: array
        integer :: i
        k = 0
        do i = 1, size (array)
           if (c == array (i)) then
              k = i
              return
           end if
        end do
      end function find
    end subroutine color_canonicalize
```

Return an array of different color indices from an array of colors. The last argument is a pseudo-color array, where the color entries correspond to the position of the corresponding index entry in the index array. The colors are assumed to be diagonal.

⟨Colors: procedures⟩+≡

```fortran
subroutine extract_color_line_indices (col, c_index, col_pos)
  type(color_t), dimension(:), intent(in) :: col
  integer, dimension(:), intent(out), allocatable :: c_index
  type(color_t), dimension(size(col)), intent(out) :: col_pos
  integer, dimension(:), allocatable :: c_tmp
  integer :: i, j, k, n, c
  allocate (c_tmp (sum (color_number (col))))
  n = 0
  SCAN1: do i = 1, size (col)
     SCAN2: do j = 1, 2
        c = abs (col(i)%c1(j))
        if (c /= 0) then
           do k = 1, n
              if (c_tmp(k) == c) then
                 col_pos(i)%c1(j) = k
                 cycle SCAN2
              end if
           end do
           n = n + 1
           c_tmp(n) = c
           col_pos(i)%c1(j) = n
        end if
     end do SCAN2
  end do SCAN1
  allocate (c_index (n))
  c_index = c_tmp(1:n)
end subroutine extract_color_line_indices
```

Given a color array, pairwise contract the color lines in all possible ways and return the resulting array of arrays. The input color array must be diagonal, and each color should occur exactly twice, once as color and once as anticolor.

Gluon entries with equal color and anticolor are explicitly excluded.

This algorithm is generic, but for long arrays it is neither efficient, nor does it avoid duplicates. It is intended for small arrays, in particular for the state matrix of a structure-function pair.

⟨*Colors: public*⟩+≡
```fortran
public :: color_array_make_contractions
```

⟨*Colors: procedures*⟩+≡
```fortran
subroutine color_array_make_contractions (col_in, col_out)
  type(color_t), dimension(:), intent(in) :: col_in
  type(color_t), dimension(:,:), intent(out), allocatable :: col_out
  type :: entry_t
     integer, dimension(:), allocatable :: map
     type(color_t), dimension(:), allocatable :: col
     type(entry_t), pointer :: next => null ()
  end type entry_t
  type :: list_t
     integer :: n = 0
     type(entry_t), pointer :: first => null ()
     type(entry_t), pointer :: last => null ()
  end type list_t
  type(list_t) :: list
  type(entry_t), pointer :: entry
```

```fortran
      integer, dimension(:), allocatable :: c_index
      type(color_t), dimension(size(col_in)) :: col_pos
      integer :: n_prt, n_c_index
      integer, dimension(:), allocatable :: map
      integer :: i, j, c
      n_prt = size (col_in)
      call extract_color_line_indices (col_in, c_index, col_pos)
      ! print *, c_index       !!! Debugging
      n_c_index = size (c_index)
      allocate (map (n_c_index))
      map = 0
      call list_append_if_valid (list, map)
      entry => list%first
      do while (associated (entry))
         do i = 1, n_c_index
            if (entry%map(i) == 0) then
               c = c_index(i)
               do j = i + 1, n_c_index
                  if (entry%map(j) == 0) then
                     map = entry%map
                     map(i) = c
                     map(j) = c
                     call list_append_if_valid (list, map)
                  end if
               end do
            end if
         end do
         entry => entry%next
      end do
      call list_to_array (list, col_out)
   contains
      subroutine list_append_if_valid (list, map)
        type(list_t), intent(inout) :: list
        integer, dimension(:), intent(in) :: map
        type(entry_t), pointer :: entry
        integer :: i, j, c, p
        entry => list%first
        do while (associated (entry))
           if (all (map == entry%map))  return
           entry => entry%next
        end do
        allocate (entry)
        allocate (entry%map (n_c_index))
        entry%map = map
        allocate (entry%col (n_prt))
        do i = 1, n_prt
           do j = 1, 2
              c = col_in(i)%c1(j)
              if (c /= 0) then
                 p = col_pos(i)%c1(j)
                 if (map(p) /= 0) then
                    entry%col(i)%c1(j) = sign (map(p), c)
                 else
                    entry%col(i)%c1(j) = c
```

```
                endif
                entry%col(i)%c2(j) = entry%col(i)%c1(j)
             end if
          end do
          if (any (entry%col(i)%c1 /= 0) .and. &
               entry%col(i)%c1(1) == - entry%col(i)%c1(2))  return
       end do
       ! call color_write (entry%col); print *, map     !!! Debugging
       if (associated (list%last)) then
          list%last%next => entry
       else
          list%first => entry
       end if
       list%last => entry
       list%n = list%n + 1
    end subroutine list_append_if_valid
    subroutine list_to_array (list, col)
      type(list_t), intent(inout) :: list
      type(color_t), dimension(:,:), intent(out), allocatable :: col
      type(entry_t), pointer :: entry
      integer :: i
      allocate (col (n_prt, list%n - 1))
      do i = 0, list%n - 1
         entry => list%first
         list%first => list%first%next
         if (i /= 0)  col(:,i) = entry%col
         deallocate (entry)
      end do
      list%last => null ()
    end subroutine list_to_array
  end subroutine color_array_make_contractions
```

Invert the color index, switching from particle to antiparticle. For gluons, we have to swap the order of color entries.

⟨*Colors: public*⟩+≡

```
    public :: color_invert
```

⟨*Colors: procedures*⟩+≡

```
    elemental subroutine color_invert (col)
      type(color_t), intent(inout) :: col
      col%c1 = - col%c1
      col%c2 = - col%c2
      if (col%c1(1) < 0 .and. col%c1(2) > 0) then
         col%c1 = col%c1(2:1:-1)
         col%c2 = col%c2(2:1:-1)
      end if
    end subroutine color_invert
```

Make a color map for two matching color arrays. The result is an array of integer pairs.

⟨*Colors: public*⟩+≡

```
    public :: set_color_map
```

⟨*Colors: procedures*⟩+≡
```
    subroutine set_color_map (map, col1, col2)
      integer, dimension(:,:), intent(out), allocatable :: map
      type(color_t), dimension(:), intent(in) :: col1, col2
      integer, dimension(:,:), allocatable :: map1
      integer :: i, j, k
      allocate (map1 (2, 2 * sum (color_number (col1))))
      k = 0
      do i = 1, size (col1)
         do j = 1, size (col1(i)%c1)
           if (col1(i)%c1(j) /= 0 &
                .and. all (map1(1,:k) /= abs (col1(i)%c1(j)))) then
              k = k + 1
              map1(1,k) = abs (col1(i)%c1(j))
              map1(2,k) = abs (col2(i)%c1(j))
           end if
           if (col1(i)%c2(j) /= 0 &
                .and. all (map1(1,:k) /= abs (col1(i)%c2(j)))) then
              k = k + 1
              map1(1,k) = abs (col1(i)%c2(j))
              map1(2,k) = abs (col2(i)%c2(j))
           end if
         end do
      end do
      allocate (map (2, k))
      map(:,:) = map1(:,:k)
    end subroutine set_color_map
```

Translate colors which have a match in the translation table (an array of integer pairs). Color that do not match an entry are simply transferred; this is done by first transferring all components, then modifiying entries where appropriate.

⟨*Colors: public*⟩+≡
```
    public :: color_translate
```

⟨*Colors: interfaces*⟩+≡
```
    interface color_translate
       module procedure color_translate0
       module procedure color_translate0_offset
       module procedure color_translate1
    end interface
```

⟨*Colors: procedures*⟩+≡
```
    subroutine color_translate0 (col, map)
      type(color_t), intent(inout) :: col
      integer, dimension(:,:), intent(in) :: map
      type(color_t) :: col_tmp
      integer :: i
      col_tmp = col
      do i = 1, size (map,2)
         where (abs (col%c1) == map(1,i))
            col_tmp%c1 = sign (map(2,i), col%c1)
         end where
         where (abs (col%c2) == map(1,i))
```

```
             col_tmp%c2 = sign (map(2,i), col%c2)
           end where
         end do
         col = col_tmp
     end subroutine color_translate0

     subroutine color_translate0_offset (col, map, offset)
       type(color_t), intent(inout) :: col
       integer, dimension(:,:), intent(in) :: map
       integer, intent(in) :: offset
       logical, dimension(size(col%c1)) :: mask1, mask2
       type(color_t) :: col_tmp
       integer :: i
       col_tmp = col
       mask1 = col%c1 /= 0
       mask2 = col%c2 /= 0
       do i = 1, size (map,2)
          where (abs (col%c1) == map(1,i))
             col_tmp%c1 = sign (map(2,i), col%c1)
             mask1 = .false.
          end where
          where (abs (col%c2) == map(1,i))
             col_tmp%c2 = sign (map(2,i), col%c2)
             mask2 = .false.
          end where
       end do
       col = col_tmp
       where (mask1)  col%c1 = sign (abs (col%c1) + offset, col%c1)
       where (mask2)  col%c2 = sign (abs (col%c2) + offset, col%c2)
     end subroutine color_translate0_offset

     subroutine color_translate1 (col, map, offset)
       type(color_t), dimension(:), intent(inout) :: col
       integer, dimension(:,:), intent(in) :: map
       integer, intent(in), optional :: offset
       integer :: i
       if (present (offset)) then
          do i = 1, size (col)
             call color_translate0_offset (col(i), map, offset)
          end do
       else
          do i = 1, size (col)
             call color_translate0 (col(i), map)
          end do
       end if
     end subroutine color_translate1
```

Merge two color objects by taking the first entry from the first and the second entry from the second argument. Makes sense only if the input colors are defined (and diagonal). If either one is undefined, transfer the defined one.

   For a color ghost, color is not defined. These have to be treated separately.

⟨*Colors: public*⟩+≡

```
     public :: operator(.merge.)
```

⟨*Colors: interfaces*⟩+≡
```
interface operator(.merge.)
   module procedure merge_colors
end interface
```

⟨*Colors: procedures*⟩+≡
```
elemental function merge_colors (col1, col2) result (col)
  type(color_t) :: col
  type(color_t), intent(in) :: col1, col2
  if (color_is_defined (col1) .and. color_is_defined (col2)) then
     call color_init_arrays (col, col1%c1, col2%c1)
  else if (color_is_defined (col1)) then
     col = col1
  else if (color_is_defined (col2)) then
     col = col2
  else if (color_is_ghost (col1)) then
     col = col1
  else if (color_is_ghost (col2)) then
     col = col2
  end if
end function merge_colors
```

Compute the color factor, given two interfering color arrays.

⟨*Colors: public*⟩+≡
```
public :: compute_color_factor
```

⟨*Colors: procedures*⟩+≡
```
function compute_color_factor (col1, col2, nc) result (factor)
  real(default) :: factor
  type(color_t), dimension(:), intent(in) :: col1, col2
  integer, intent(in), optional :: nc
  type(color_t), dimension(size(col1)) :: col
  integer :: ncol, nloops, nghost
  ncol = 3;  if (present (nc))  ncol = nc
  col = col1 .merge. col2
  nloops = count_color_loops (col)
  nghost = count (color_is_ghost (col))
  factor = real (ncol, default) ** (nloops - nghost)
  if (color_ghost_parity (col))  factor = - factor
end function compute_color_factor
```

We have a pair of color index arrays which corresponds to a squared matrix element. We want to determine the number of color loops in this square matrix element. So we first copy the colors (stored in a single color array with a pair of color lists in each entry) to a temporary where the color indices are shifted by some offset. We then recursively follow each loop, starting at the first color that has the offset, resetting the first color index to the loop index and each further index to zero as we go. We check that (a) each color index occurs twice within the left (right) color array, (b) the loops are closed, so we always come back to a line which has the loop index.

In order for the algorithm to work we have to conjugate the colors of initial state particles (one for decays, two for scatterings) into their corresponding

anticolors of outgoing particles.

⟨*Colors: public*⟩+≡
```
    public :: count_color_loops
```

⟨*Colors: procedures*⟩+≡
```
    function count_color_loops (col) result (count)
      integer :: count
      type(color_t), dimension(:), intent(in) :: col
      type(color_t), dimension(size(col)) :: cc
      integer :: i, n, offset
      ! print *, "Count color loops:"      !!! Debugging
      ! call color_write (col); print *    !!! Debugging
      cc = col
      n = size (cc)
      offset = n
      call color_add_offset (cc, offset)
      ! print *, offset                    !!! Debugging
      ! call color_write (cc); print *     !!! Debugging
      count = 0
      SCAN_LOOPS: do
         do i = 1, n
            ! print *, i, ':', cc(i)%c1    !!! Debugging
            if (color_is_defined (cc(i))) then
               if (any (cc(i)%c1 > offset)) then
                  ! print *, 'start', i     !!! Debugging
                  count = count + 1
                  call follow_line1 (pick_new_line (cc(i)%c1, count, 1))
                  cycle SCAN_LOOPS
               end if
            end if
         end do
         exit SCAN_LOOPS
      end do SCAN_LOOPS
    contains
      function pick_new_line (c, reset_val, sgn) result (line)
        integer :: line
        integer, dimension(:), intent(inout) :: c
        integer, intent(in) :: reset_val
        integer, intent(in) :: sgn
        integer :: i
        if (any (c == count)) then
           line = count
        else
           do i = 1, size (c)
              if (sign (1, c(i)) == sgn .and. abs (c(i)) > offset) then
                 line = c(i)
                 c(i) = reset_val
                 return
              end if
           end do
           call color_mismatch
        end if
      end function pick_new_line
      subroutine reset_line (c, line)
        integer, dimension(:), intent(inout) :: c
```

```
      integer, intent(in) :: line
      integer :: i
      do i = 1, size (c)
         if (c(i) == line) then
            c(i) = 0
            return
         end if
      end do
   end subroutine reset_line
   recursive subroutine follow_line1 (line)
      integer, intent(in) :: line
      integer :: i
      ! print *, 'follow line 1:', line      !!! Debugging
      if (line == count) then
         ! print *, 'loop closed'            !!! Debugging
         return
      end if
      do i = 1, n
         if (any (cc(i)%c1 == -line)) then
            call reset_line (cc(i)%c1, -line)
            ! print *, 'found', -line, ' resetting c1:'   !!! Debugging
            ! call color_write (cc); print *             !!! Debugging
            call follow_line2 (pick_new_line (cc(i)%c2, 0, sign (1, -line)))
            return
         end if
      end do
      call color_mismatch ()
   end subroutine follow_line1
   recursive subroutine follow_line2 (line)
      integer, intent(in) :: line
      integer :: i
      ! print *, 'follow line 2:', line      !!! Debugging
      do i = 1, n
         if (any (cc(i)%c2 == -line)) then
            call reset_line (cc(i)%c2, -line)
            ! print *, 'found', -line, ' resetting c2:'   !!! Debugging
            ! call color_write (cc); print *             !!! Debugging
            call follow_line1 (pick_new_line (cc(i)%c1, 0, sign (1, -line)))
            return
         end if
      end do
      call color_mismatch ()
   end subroutine follow_line2
   subroutine color_mismatch ()
      call color_write (col)
      print *
      call msg_bug (" Color flow mismatch (color loops should be closed)")
   end subroutine color_mismatch
end function count_color_loops
```

7.2.6 Unit tests

⟨*Colors: public*⟩+≡

```
      public :: color_test
```
⟨*Colors: procedures*⟩+≡
```
    subroutine color_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Colors: execute tests⟩
      end subroutine color_test
```

This is a color counting test.

⟨*Colors: execute tests*⟩≡
```
    call test (color_1, "color_1", &
        "check color counting", &
        u, results)
```
⟨*Colors: tests*⟩≡
```
    subroutine color_1 (u)
      integer, intent(in) :: u
      type(color_t), dimension(4) :: col1, col2, col
      type(color_t), dimension(:), allocatable :: col3
      type(color_t), dimension(:,:), allocatable :: col_array
      integer :: count, i
      call color_init_col_acl (col1, [1, 0, 2, 3], [0, 1, 3, 2])
      col2 = col1
      call color_write (col1, u)
      write (u, "(A)")
      call color_write (col2, u)
      write (u, "(A)")
      col = col1 .merge. col2
      call color_write (col, u)
      write (u, "(A)")
      count = count_color_loops (col)
      write (u, "(A,I1)") "Number of color loops (3): ", count
      call color_init_col_acl (col2, [1, 0, 2, 3], [0, 2, 3, 1])
      call color_write (col1, u)
      write (u, "(A)")
      call color_write (col2, u)
      write (u, "(A)")
      col = col1 .merge. col2
      call color_write (col, u)
      write (u, "(A)")
      count = count_color_loops (col)
      write (u, "(A,I1)")  "Number of color loops (2): ", count
      write (u, "(A)")
      allocate (col3 (4))
      call color_init_from_array (col3, &
          reshape ([1, 0,   0, -1,  2, -3,  3, -2], &
                    [2, 4]))
      call color_write (col3, u)
      write (u, "(A)")
      call color_array_make_contractions (col3, col_array)
      write (u, "(A)")  "Contractions:"
      do i = 1, size (col_array, 2)
         call color_write (col_array(:,i), u)
         write (u, "(A)")
```

```
   end do
   deallocate (col3)
   write (u, "(A)")
   allocate (col3 (6))
   call color_init_from_array (col3, &
        reshape ([1, -2,   3, 0,  0, -1,  2, -4,  -3, 0,  4, 0], &
                 [2, 6]))
   call color_write (col3, u)
   write (u, "(A)")
   call color_array_make_contractions (col3, col_array)
   write (u, "(A)")  "Contractions:"
   do i = 1, size (col_array, 2)
      call color_write (col_array(:,i), u)
      write (u, "(A)")
   end do
 end subroutine color_1
```

7.2.7 The Madgraph color model

This section describes the method for matrix element and color flow calculation within Madgraph.

For each Feynman diagram, the colorless amplitude for a specified helicity and momentum configuration (in- and out- combined) is computed:

$$A_d(p, h) \tag{7.1}$$

Inserting color, the squared matrix element for definite helicity and momentum is

$$M^2(p, h) = \sum_{dd'} A_d(p, h) \, C_{dd'} A_{d'}^*(p, h) \tag{7.2}$$

where $C_{dd'}$ describes the color interference of the two diagrams A_d and A'_d, which is independent of momentum and helicity and can be calculated for each Feynman diagram pair by reducing it to the corresponding color graph. Obviously, one could combine all diagrams with identical color structure, such that the index d runs only over different color graphs. For colorless diagrams all elements of $C_{dd'}$ are equal to unity.

The hermitian matrix $C_{dd'}$ is diagonalized once and for all, such that it can be written in the form

$$C_{dd'} = \sum_\lambda c_d^\lambda \lambda \, c_d^{\lambda *}, \tag{7.3}$$

where the eigenvectors c_d are normalized,

$$\sum_d |c_d^\lambda|^2 = 1, \tag{7.4}$$

and the λ values are the corresponding eigenvalues. In the colorless case, this means $c_d = 1/\sqrt{N_d}$ for all diagrams (N_d = number of diagrams), and $\lambda = N_d$ is the only nonzero eigenvalue.

Consequently, the squared matrix element for definite helicity and momentum can also be written as

$$M^2(p, h) = \sum_\lambda A_\lambda(p, h) \, \lambda \, A_\lambda(p, h)^* \tag{7.5}$$

with

$$A_\lambda(p, h) = \sum_d c_d^\lambda A_d(p, h). \tag{7.6}$$

For generic spin density matrices, this is easily generalized to

$$M^2(p, h, h') = \sum_\lambda A_\lambda(p, h)\, \lambda\, A_\lambda(p, h')^* \tag{7.7}$$

To determine the color flow probabilities of a given momentum-helicity configuration, the color flow amplitudes are calculated as

$$a_f(p, h) = \sum_d \beta_d^f A_d(p, h), \tag{7.8}$$

where the coefficients β_d^f describe the amplitude for a given Feynman diagram (or color graph) d to correspond to a definite color flow f. They are computed from $C_{dd'}$ by transforming this matrix into the color flow basis and neglecting all off-diagonal elements. Again, these coefficients do not depend on momentum or helicity and can therefore be calculated in advance. This gives the color flow transition matrix

$$F^f(p, h, h') = a_f(p, h)\, a_f^*(p, h') \tag{7.9}$$

which is assumed diagonal in color flow space and is separate from the color-summed transition matrix M^2. They are, however, equivalent (up to a factor) to leading order in $1/N_c$, and using the color flow transition matrix is appropriate for matching to hadronization.

Note that the color flow transition matrix is not normalized at this stage. To make use of it, we have to fold it with the in-state density matrix to get a pseudo density matrix

$$\hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h_{\text{out}}') = \sum_{h_{\text{in}} h_{\text{in}}'} F^f(p, h, h')\, \rho_{\text{in}}(p, h_{\text{in}}, h_{\text{in}}') \tag{7.10}$$

which gets a meaning only after contracted with projections on the outgoing helicity states k_{out}, given as linear combinations of helicity states with the unitary coefficient matrix $c(k_{\text{out}}, h_{\text{out}})$. Then the probability of finding color flow f when the helicity state k_{out} is measured is given by

$$P^f(p, k_{\text{out}}) = Q^f(p, k_{\text{out}}) / \sum_f Q^f(p, k_{\text{out}}) \tag{7.11}$$

where

$$Q^f(p, k_{\text{out}}) = \sum_{h_{\text{out}} h_{\text{out}}'} c(k_{\text{out}}, h_{\text{out}})\, \hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h_{\text{out}}')\, c^*(k_{\text{out}}, h_{\text{out}}') \tag{7.12}$$

However, if we can assume that the out-state helicity basis is the canonical one, we can throw away the off diagonal elements in the color flow density matrix and normalize the ones on the diagonal to obtain

$$P^f(p, h_{\text{out}}) = \hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h_{\text{out}}) / \sum_f \hat{\rho}_{\text{out}}^f(p, h_{\text{out}}, h_{\text{out}}) \tag{7.13}$$

Finally, the color-summed out-state density matrix is computed by the scattering formula

$$\rho_{\text{out}}(p, h_{\text{out}}, h'_{\text{out}}) = \sum_{h_{\text{in}} h'_{\text{in}}} M^2(p, h, h') \, \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}) \tag{7.14}$$

$$= \sum_{h_{\text{in}} h'_{\text{in}} \lambda} A_\lambda(p, h) \, \lambda \, A_\lambda(p, h')^* \rho_{\text{in}}(p, h_{\text{in}}, h'_{\text{in}}), \tag{7.15}$$

The trace of ρ_{out} is the squared matrix element, summed over all internal degrees of freedom. To get the squared matrix element for a definite helicity k_{out} and color flow f, one has to project the density matrix onto the given helicity state and multiply with $P^f(p, k_{\text{out}})$.

For diagonal helicities the out-state density reduces to

$$\rho_{\text{out}}(p, h_{\text{out}}) = \sum_{h_{\text{in}} \lambda} \lambda |A_\lambda(p, h)|^2 \rho_{\text{in}}(p, h_{\text{in}}). \tag{7.16}$$

Since no basis transformation is involved, we can use the normalized color flow probability $P^f(p, h_{\text{out}})$ and express the result as

$$\rho_{\text{out}}^f(p, h_{\text{out}}) = \rho_{\text{out}}(p, h_{\text{out}}) \, P^f(p, h_{\text{out}}) \tag{7.17}$$

$$= \sum_{h_{\text{in}} \lambda} \frac{|a^f(p, h)|^2}{\sum_f |a^f(p, h)|^2} \lambda |A_\lambda(p, h)|^2 \rho_{\text{in}}(p, h_{\text{in}}). \tag{7.18}$$

From these considerations, the following calculation strategy can be derived:

- Before the first event is generated, the color interference matrix $C_{dd'}$ is computed and diagonalized, so the eigenvectors c_d^λ, eigenvalues λ and color flow coefficients β_d^f are obtained. In practice, these calculations are done when the matrix element code is generated, and the results are hardcoded in the matrix element subroutine as `DATA` statements.

- For each event, one loops over helicities once and stores the matrices $A_\lambda(p, h)$ and $a^f(p, h)$. The allowed color flows, helicity combinations and eigenvalues are each labeled by integer indices, so one has to store complex matrices of dimension $N_\lambda \times N_h$ and $N_f \times N_h$, respectively.

- The further strategy depends on the requested information.

  1. If colorless diagonal helicity amplitudes are required, the eigenvalues $A_\lambda(p, h)$ are squared, summed with weight λ, and the result contracted with the in-state probability vector $\rho_{\text{in}}(p, h_{\text{in}})$. The result is a probability vector $\rho_{\text{out}}(p, h_{\text{out}})$.

  2. For colored diagonal helicity amplitudes, the color coefficients $a^f(p, h)$ are also squared and used as weights to obtain the color-flow probability vector $\rho_{\text{out}}^f(p, h_{\text{out}})$.

  3. For colorless non-diagonal helicity amplitudes, we contract the tensor product of $A_\lambda(p, h)$ with $A_\lambda(p, h')$, weighted with λ, with the correlated in-state density matrix, to obtain a correlated out-state density matrix.

713

4. In the general (colored, non-diagonal) case, we do the same as in the colorless case, but return the un-normalized color flow density matrix $\hat{\rho}^f_{\text{out}}(p, h_{\text{out}}, h'_{\text{out}})$ in addition. When the relevant helicity basis is known, the latter can be used by the caller program to determine flow probabilities. (In reality, we assume the canonical basis and reduce the correlated out-state density to its diagonal immediately.)

7.3 Flavors: Particle properties

This module contains a type for holding the flavor code, and all functions that depend on the model, i.e., that determine particle properties.

The PDG code is packed in a special `flavor` type. (This prohibits meaningless operations, and it allows for a different implementation, e.g., some non-PDG scheme internally, if appropiate at some point.) In addition, the flavor object holds a pointer to a `particle_data` object which is centrally stored (as part of a physics model). In this way, all particle data can be accessed using just the `flv` object without having to carry them around.

The pointer component imposes a technical restriction: Assignment of flavor objects cannot be used, directly or indirectly, in pure, and thus in elemental procedures.

⟨`flavors.f90`⟩≡
  ⟨*File header*⟩

  ```
  module flavors
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use pdg_arrays
    use colors, only: color_t, color_init
    use models
```

  ⟨*Standard module head*⟩

  ⟨*Flavors: public*⟩

  ⟨*Flavors: types*⟩

  ⟨*Flavors: interfaces*⟩

  ```
  contains
```

  ⟨*Flavors: procedures*⟩

  ```
  end module flavors
```

7.3.1 The flavor type

The flavor type is an integer representing the PDG code, or undefined (zero). Negative codes represent antiflavors.

For full generality, and analogy with helicity and color, we allow for non-diagonal flavor indices in density matrices. This is probably academic; an obscure application is the definition of proper isospin states.

Further properties of the given flavor can be retrieved via the particle-data pointer, if it is associated.

⟨*Flavors: public*⟩≡
  ```
    public :: flavor_t
```

⟨*Flavors: types*⟩≡

```
type :: flavor_t
   private
   integer :: f = UNDEFINED
   type(particle_data_t), pointer :: prt => null ()
end type flavor_t
```

Initializer form. If the model is assigned, the procedure is impure, therefore we have to define a separate array version.

⟨*Flavors: public*⟩+≡

```
public :: flavor_init
```

⟨*Flavors: interfaces*⟩≡

```
interface flavor_init
   module procedure flavor_init0_empty
   module procedure flavor_init0
   module procedure flavor_init0_particle_data
   module procedure flavor_init0_model
   module procedure flavor_init0_name_model
   module procedure flavor_init1_model
   module procedure flavor_init1_name_model
   module procedure flavor_init2_model
   module procedure flavor_init_aval_model
end interface
```

⟨*Flavors: procedures*⟩≡

```
elemental subroutine flavor_init0_empty (flv)
  type(flavor_t), intent(out) :: flv
end subroutine flavor_init0_empty

elemental subroutine flavor_init0 (flv, f)
  type(flavor_t), intent(out) :: flv
  integer, intent(in) :: f
  flv%f = f
end subroutine flavor_init0

subroutine flavor_init0_particle_data (flv, particle_data)
  type(flavor_t), intent(out) :: flv
  type(particle_data_t), intent(in), target :: particle_data
  flv%f = particle_data_get_pdg (particle_data)
  flv%prt => particle_data
end subroutine flavor_init0_particle_data

subroutine flavor_init0_model (flv, f, model)
  type(flavor_t), intent(out) :: flv
  integer, intent(in) :: f
  type(model_t), intent(in), target :: model
  flv%f = f
  flv%prt => model_get_particle_ptr (model, f)
end subroutine flavor_init0_model

subroutine flavor_init1_model (flv, f, model)
  type(flavor_t), dimension(:), intent(out) :: flv
  integer, dimension(:), intent(in) :: f
  type(model_t), intent(in), target :: model
```

716

```
      integer :: i
      do i = 1, size (f)
         call flavor_init0_model (flv(i), f(i), model)
      end do
   end subroutine flavor_init1_model

   subroutine flavor_init2_model (flv, f, model)
      type(flavor_t), dimension(:,:), intent(out) :: flv
      integer, dimension(:,:), intent(in) :: f
      type(model_t), intent(in), target :: model
      integer :: i
      do i = 1, size (f, 2)
         call flavor_init1_model (flv(:,i), f(:,i), model)
      end do
   end subroutine flavor_init2_model

   subroutine flavor_init0_name_model (flv, name, model)
      type(flavor_t), intent(out) :: flv
      type(string_t), intent(in) :: name
      type(model_t), intent(in), target :: model
      flv%f = model_get_particle_pdg (model, name)
      flv%prt => model_get_particle_ptr (model, flv%f)
   end subroutine flavor_init0_name_model

   subroutine flavor_init1_name_model (flv, name, model)
      type(flavor_t), dimension(:), intent(out) :: flv
      type(string_t), dimension(:), intent(in) :: name
      type(model_t), intent(in), target :: model
      integer :: i
      do i = 1, size (name)
         call flavor_init0_name_model (flv(i), name(i), model)
      end do
   end subroutine flavor_init1_name_model
```

This version transforms a PDG array value into a flavor array. The flavor array
must be allocatable.

⟨Flavors: procedures⟩+≡
```
   subroutine flavor_init_aval_model (flv, aval, model)
      type(flavor_t), dimension(:), intent(out), allocatable :: flv
      type(pdg_array_t), intent(in) :: aval
      type(model_t), intent(in), target :: model
      integer, dimension(:), allocatable :: pdg
      pdg = aval
      allocate (flv (size (pdg)))
      call flavor_init (flv, pdg, model)
   end subroutine flavor_init_aval_model
```

Undefine the flavor state:

⟨Flavors: public⟩+≡
```
   public :: flavor_undefine
```

⟨Flavors: procedures⟩+≡
```
   elemental subroutine flavor_undefine (flv)
```

```
      type(flavor_t), intent(inout) :: flv
      flv%f = UNDEFINED
      flv%prt => null ()
   end subroutine flavor_undefine
```

Output: dense, no linebreak

⟨*Flavors: public*⟩+≡
```
    public :: flavor_write
```

⟨*Flavors: procedures*⟩+≡
```
    subroutine flavor_write (flv, unit)
      type(flavor_t), intent(in) :: flv
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (associated (flv%prt)) then
         write (u, "(A)", advance="no")  "f("
      else
         write (u, "(A)", advance="no")  "p("
      end if
      write (u, "(I0)", advance="no")  flv%f
      write (u, "(A)", advance="no")  ")"
    end subroutine flavor_write
```

Binary I/O. Currently, the model information is not written/read, so after reading the particle-data pointer is empty.

⟨*Flavors: public*⟩+≡
```
    public :: flavor_write_raw
    public :: flavor_read_raw
```

⟨*Flavors: procedures*⟩+≡
```
    subroutine flavor_write_raw (flv, u)
      type(flavor_t), intent(in) :: flv
      integer, intent(in) :: u
      write (u) flv%f
    end subroutine flavor_write_raw

    subroutine flavor_read_raw (flv, u, iostat)
      type(flavor_t), intent(out) :: flv
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) flv%f
    end subroutine flavor_read_raw
```

Assignment

Default assignment of flavor objects is possible, but cannot be used in pure procedures, because a pointer assignment is involved.

Assign the particle pointer separately. This cannot be elemental, so we define a scalar and an array version explicitly. We refer to an array of flavors, not an array of models.

⟨*Flavors: public*⟩+≡

```
    public :: flavor_set_model
```

⟨*Flavors: interfaces*⟩+≡
```
    interface flavor_set_model
       module procedure flavor_set_model_single
       module procedure flavor_set_model_array
       module procedure flavor_set_model_array2
    end interface
```

⟨*Flavors: procedures*⟩+≡
```
    subroutine flavor_set_model_single (flv, model)
      type(flavor_t), intent(inout) :: flv
      type(model_t), intent(in), target :: model
      if (flv%f /= UNDEFINED) &
           flv%prt => model_get_particle_ptr (model, flv%f)
    end subroutine flavor_set_model_single

    subroutine flavor_set_model_array (flv, model)
      type(flavor_t), dimension(:), intent(inout) :: flv
      type(model_t), intent(in), target :: model
      integer :: i
      do i = 1, size (flv)
         if (flv(i)%f /= UNDEFINED) &
              flv(i)%prt => model_get_particle_ptr (model, flv(i)%f)
      end do
    end subroutine flavor_set_model_array

    subroutine flavor_set_model_array2 (flv, model)
      type(flavor_t), dimension(:,:), intent(inout) :: flv
      type(model_t), intent(in), target :: model
      integer :: i, j
      do i = 1, size (flv, 2)
         do j = 1, size (flv, 1)
            if (flv(j,i)%f /= UNDEFINED) &
                 flv(j,i)%prt => model_get_particle_ptr (model, flv(j,i)%f)
         end do
      end do
    end subroutine flavor_set_model_array2
```

Predicates

Return the definition status

⟨*Flavors: public*⟩+≡
```
    public :: flavor_is_defined
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_is_defined (flv) result (defined)
      logical :: defined
      type(flavor_t), intent(in) :: flv
      defined = flv%f /= UNDEFINED
    end function flavor_is_defined
```

719

Check for valid flavor (including undefined):

⟨*Flavors: public*⟩+≡
```
    public :: flavor_is_valid
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_is_valid (flv) result (valid)
      logical :: valid
      type(flavor_t), intent(in) :: flv
      valid = flv%f /= INVALID
    end function flavor_is_valid
```

Return true if the particle-data pointer is associated. (Debugging aid)

⟨*Flavors: public*⟩+≡
```
    public :: flavor_is_associated
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_is_associated (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      flag = associated (flv%prt)
    end function flavor_is_associated
```

Accessing contents

With the exception of the PDG code, all particle properties are accessible only via the `prt` pointer. If this is unassigned, some access function will crash.

Return the flavor as an integer

⟨*Flavors: public*⟩+≡
```
    public :: flavor_get_pdg
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_get_pdg (flv) result (f)
      integer :: f
      type(flavor_t), intent(in) :: flv
      f = flv%f
    end function flavor_get_pdg
```

Return the flavor of the antiparticle

⟨*Flavors: public*⟩+≡
```
    public :: flavor_get_pdg_anti
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_get_pdg_anti (flv) result (f)
      integer :: f
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         if (particle_data_has_antiparticle (flv%prt)) then
            f = -flv%f
         else
            f = flv%f
         end if
      else
         f = 0
```

```
      end if
    end function flavor_get_pdg_anti
```

Absolute value:

⟨*Flavors: public*⟩+≡
```
    public :: flavor_get_pdg_abs
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_get_pdg_abs (flv) result (f)
      integer :: f
      type(flavor_t), intent(in) :: flv
      f = abs (flv%f)
    end function flavor_get_pdg_abs
```

Generic properties

⟨*Flavors: public*⟩+≡
```
    public :: flavor_is_visible
    public :: flavor_is_parton
    public :: flavor_is_beam_remnant
    public :: flavor_is_gauge
    public :: flavor_is_left_handed
    public :: flavor_is_right_handed
    public :: flavor_is_antiparticle
    public :: flavor_has_antiparticle
    public :: flavor_is_stable
    public :: flavor_get_decays
    public :: flavor_decays_isotropically
    public :: flavor_decays_diagonal
    public :: flavor_is_polarized
```

⟨*Flavors: procedures*⟩+≡
```
    elemental function flavor_is_visible (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         flag = particle_data_is_visible (flv%prt)
      else
         flag = .false.
      end if
    end function flavor_is_visible

    elemental function flavor_is_parton (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
         flag = particle_data_is_parton (flv%prt)
      else
         flag = .false.
      end if
    end function flavor_is_parton

    elemental function flavor_is_beam_remnant (flv) result (flag)
      logical :: flag
      type(flavor_t), intent(in) :: flv
```

```
    select case (abs (flv%f))
    case (HADRON_REMNANT, &
          HADRON_REMNANT_SINGLET, HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET)
       flag = .true.
    case default
       flag = .false.
    end select
end function flavor_is_beam_remnant

elemental function flavor_is_gauge (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
     flag = particle_data_is_gauge (flv%prt)
  else
     flag = .false.
  end if
end function flavor_is_gauge

elemental function flavor_is_left_handed (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
     if (flv%f > 0) then
        flag = particle_data_is_left_handed (flv%prt)
     else
        flag = particle_data_is_right_handed (flv%prt)
     end if
  else
     flag = .false.
  end if
end function flavor_is_left_handed

elemental function flavor_is_right_handed (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  if (associated (flv%prt)) then
     if (flv%f > 0) then
        flag = particle_data_is_right_handed (flv%prt)
     else
        flag = particle_data_is_left_handed (flv%prt)
     end if
  else
     flag = .false.
  end if
end function flavor_is_right_handed

elemental function flavor_is_antiparticle (flv) result (flag)
  logical :: flag
  type(flavor_t), intent(in) :: flv
  flag = flv%f < 0
end function flavor_is_antiparticle

elemental function flavor_has_antiparticle (flv) result (flag)
```

```fortran
    logical :: flag
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
       flag = particle_data_has_antiparticle (flv%prt)
    else
       flag = .false.
    end if
  end function flavor_has_antiparticle

  elemental function flavor_is_stable (flv) result (flag)
    logical :: flag
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
       flag = particle_data_is_stable (flv%prt, anti = flv%f < 0)
    else
       flag = .true.
    end if
  end function flavor_is_stable

  subroutine flavor_get_decays (flv, decay)
    type(flavor_t), intent(in) :: flv
    type(string_t), dimension(:), intent(out), allocatable :: decay
    logical :: anti
    anti = flv%f < 0
    if (.not. particle_data_is_stable (flv%prt, anti)) then
       call particle_data_get_decays (flv%prt, decay, anti)
    end if
  end subroutine flavor_get_decays

  elemental function flavor_decays_isotropically (flv) result (flag)
    logical :: flag
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
       flag = particle_data_decays_isotropically (flv%prt, anti = flv%f < 0)
    else
       flag = .true.
    end if
  end function flavor_decays_isotropically

  elemental function flavor_decays_diagonal (flv) result (flag)
    logical :: flag
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
       flag = particle_data_decays_diagonal (flv%prt, anti = flv%f < 0)
    else
       flag = .true.
    end if
  end function flavor_decays_diagonal

  elemental function flavor_is_polarized (flv) result (flag)
    logical :: flag
    type(flavor_t), intent(in) :: flv
    if (associated (flv%prt)) then
       flag = particle_data_is_polarized (flv%prt, anti = flv%f < 0)
```

```
      else
         flag = .false.
      end if
   end function flavor_is_polarized
```

Names:

⟨*Flavors: public*⟩+≡
```
   public :: flavor_get_name
   public :: flavor_get_tex_name
```

⟨*Flavors: procedures*⟩+≡
```
   elemental function flavor_get_name (flv) result (name)
     type(string_t) :: name
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        name = particle_data_get_name (flv%prt, flv%f < 0)
     else
        name = "?"
     end if
   end function flavor_get_name

   elemental function flavor_get_tex_name (flv) result (name)
     type(string_t) :: name
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        name = particle_data_get_tex_name (flv%prt, flv%f < 0)
     else
        name = "?"
     end if
   end function flavor_get_tex_name
```

⟨*Flavors: public*⟩+≡
```
   public :: flavor_get_spin_type
   public :: flavor_get_multiplicity
   public :: flavor_get_isospin_type
   public :: flavor_get_charge_type
   public :: flavor_get_color_type
```

⟨*Flavors: procedures*⟩+≡
```
   elemental function flavor_get_spin_type (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        type = particle_data_get_spin_type (flv%prt)
     else
        type = 1
     end if
   end function flavor_get_spin_type

   elemental function flavor_get_multiplicity (flv) result (type)
     integer :: type
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        type = particle_data_get_multiplicity (flv%prt)
```

```
      else
          type = 1
      end if
   end function flavor_get_multiplicity

   elemental function flavor_get_isospin_type (flv) result (type)
      integer :: type
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
          type = particle_data_get_isospin_type (flv%prt)
      else
          type = 1
      end if
   end function flavor_get_isospin_type

   elemental function flavor_get_charge_type (flv) result (type)
      integer :: type
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
          type = particle_data_get_charge_type (flv%prt)
      else
          type = 1
      end if
   end function flavor_get_charge_type

   elemental function flavor_get_color_type (flv) result (type)
      integer :: type
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
          if (flavor_is_antiparticle (flv)) then
             type = - particle_data_get_color_type (flv%prt)
          else
             type = particle_data_get_color_type (flv%prt)
          end if
      else
          type = 1
      end if
   end function flavor_get_color_type
```

These functions return real values:

⟨*Flavors: public*⟩+≡

```
   public :: flavor_get_charge
   public :: flavor_get_mass
   public :: flavor_get_width
   public :: flavor_get_isospin
```

⟨*Flavors: procedures*⟩+≡

```
   elemental function flavor_get_charge (flv) result (charge)
      real(default) :: charge
      type(flavor_t), intent(in) :: flv
      if (associated (flv%prt)) then
          if (flavor_is_antiparticle (flv)) then
             charge = particle_data_get_charge (flv%prt)
          else
```

```
            charge = - particle_data_get_charge (flv%prt)
         end if
      else
         charge = 0
      end if
   end function flavor_get_charge

   elemental function flavor_get_mass (flv) result (mass)
     real(default) :: mass
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        mass = particle_data_get_mass (flv%prt)
     else
        mass = 0
     end if
   end function flavor_get_mass

   elemental function flavor_get_width (flv) result (width)
     real(default) :: width
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        width = particle_data_get_width (flv%prt)
     else
        width = 0
     end if
   end function flavor_get_width

   elemental function flavor_get_isospin (flv) result (isospin)
     real(default) :: isospin
     type(flavor_t), intent(in) :: flv
     if (associated (flv%prt)) then
        if (flavor_is_antiparticle (flv)) then
           isospin = particle_data_get_isospin (flv%prt)
        else
           isospin = - particle_data_get_isospin (flv%prt)
        end if
     else
        isospin = 0
     end if
   end function flavor_get_isospin
```

Comparisons

If one of the flavors is undefined, the other defined, they match.

⟨*Flavors: public*⟩+≡
```
    public :: operator(.match.)
    public :: operator(==)
    public :: operator(/=)
```

⟨*Flavors: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure flavor_match
    end interface
```

```
    interface operator(==)
       module procedure flavor_eq
    end interface
    interface operator(/=)
       module procedure flavor_neq
    end interface
```

⟨*Flavors: procedures*⟩+≡

```
    elemental function flavor_match (flv1, flv2) result (eq)
      logical :: eq
      type(flavor_t), intent(in) :: flv1, flv2
      if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
         eq = flv1%f == flv2%f
      else
         eq = .true.
      end if
    end function flavor_match

    elemental function flavor_eq (flv1, flv2) result (eq)
      logical :: eq
      type(flavor_t), intent(in) :: flv1, flv2
      if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
         eq = flv1%f == flv2%f
      else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
         eq = .true.
      else
         eq = .false.
      end if
    end function flavor_eq
```

⟨*Flavors: procedures*⟩+≡

```
    elemental function flavor_neq (flv1, flv2) result (neq)
      logical :: neq
      type(flavor_t), intent(in) :: flv1, flv2
      if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
         neq = flv1%f /= flv2%f
      else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
         neq = .false.
      else
         neq = .true.
      end if
    end function flavor_neq
```

Tools

Merge two flavor indices. This works only if both are equal or either one is
undefined, because we have no off-diagonal flavor entries. Otherwise, generate
an invalid flavor.

We cannot use elemental procedures because of the pointer component.

⟨*Flavors: public*⟩+≡

```
    public :: operator(.merge.)
```

⟨*Flavors: interfaces*⟩+≡
```
    interface operator(.merge.)
       module procedure merge_flavors0
       module procedure merge_flavors1
    end interface
```

⟨*Flavors: procedures*⟩+≡
```
    function merge_flavors0 (flv1, flv2) result (flv)
      type(flavor_t) :: flv
      type(flavor_t), intent(in) :: flv1, flv2
      if (flavor_is_defined (flv1) .and. flavor_is_defined (flv2)) then
         if (flv1 == flv2) then
            flv = flv1
         else
            flv%f = INVALID
         end if
      else if (flavor_is_defined (flv1)) then
         flv = flv1
      else if (flavor_is_defined (flv2)) then
         flv = flv2
      end if
    end function merge_flavors0

    function merge_flavors1 (flv1, flv2) result (flv)
      type(flavor_t), dimension(:), intent(in) :: flv1, flv2
      type(flavor_t), dimension(size(flv1)) :: flv
      integer :: i
      do i = 1, size (flv1)
         flv(i) = flv1(i) .merge. flv2(i)
      end do
    end function merge_flavors1
```

Generate consecutive color indices for a given flavor. The indices are counted starting with the stored value of c, so new indices are created each time this (impure) function is called. The counter can be reset by the optional argument `c_seed` if desired. The optional flag **reverse** is used only for octets. If set, the color and anticolor entries of the octet particle are exchanged.

⟨*Flavors: public*⟩+≡
```
    public :: color_from_flavor
```

⟨*Flavors: interfaces*⟩+≡
```
    interface color_from_flavor
       module procedure color_from_flavor0
       module procedure color_from_flavor1
    end interface
```

⟨*Flavors: procedures*⟩+≡
```
    function color_from_flavor0 (flv, c_seed, reverse) result (col)
      type(color_t) :: col
      type(flavor_t), intent(in) :: flv
      integer, intent(in), optional :: c_seed
      logical, intent(in), optional :: reverse
      integer, save :: c = 1
      logical :: rev
```

```
      if (present (c_seed))  c = c_seed
      rev = .false.;  if (present (reverse)) rev = reverse
      select case (flavor_get_color_type (flv))
      case (1)
      case (3)
         call color_init (col, (/ c /));  c = c + 1
      case (-3)
         call color_init (col, (/-c /));  c = c + 1
      case (8)
         if (rev) then
            call color_init (col, (/ c+1, -c /));  c = c + 2
         else
            call color_init (col, (/ c, -(c+1) /));  c = c + 2
         end if
      end select
   end function color_from_flavor0

   function color_from_flavor1 (flv, c_seed, reverse) result (col)
      type(flavor_t), dimension(:), intent(in) :: flv
      integer, intent(in), optional :: c_seed
      logical, intent(in), optional :: reverse
      type(color_t), dimension(size(flv)) :: col
      integer :: i
      col(1) = color_from_flavor0 (flv(1), c_seed, reverse)
      do i = 2, size (flv)
         col(i) = color_from_flavor0 (flv(i), reverse=reverse)
      end do
   end function color_from_flavor1
```

This procedure returns the flavor object for the antiparticle. The antiparticle code may either be the same code or its negative.

⟨*Flavors: public*⟩+≡
```
   public :: flavor_anti
```

⟨*Flavors: procedures*⟩+≡
```
   function flavor_anti (flv) result (aflv)
     type(flavor_t) :: aflv
     type(flavor_t), intent(in) :: flv
     if (flavor_has_antiparticle (flv)) then
        aflv%f = - flv%f
     else
        aflv%f = flv%f
     end if
     aflv%prt => flv%prt
   end function flavor_anti
```

7.4 Quantum numbers

This module collects helicity, color, and flavor in a single type and defines procedures

⟨`quantum_numbers.f90`⟩≡
  ⟨*File header*⟩

```
module quantum_numbers
```

  ⟨*Use file utils*⟩
```
    use models
    use flavors
    use colors
    use helicities
```

  ⟨*Standard module head*⟩

  ⟨*Quantum numbers: public*⟩

  ⟨*Quantum numbers: types*⟩

  ⟨*Quantum numbers: interfaces*⟩

```
  contains
```

  ⟨*Quantum numbers: procedures*⟩

```
  end module quantum_numbers
```

7.4.1 The quantum number type

⟨*Quantum numbers: public*⟩≡
```
    public :: quantum_numbers_t
```

⟨*Quantum numbers: types*⟩≡
```
    type :: quantum_numbers_t
       private
       type(flavor_t) :: f
       type(color_t) :: c
       type(helicity_t) :: h
    end type quantum_numbers_t
```

Define quantum numbers: Initializer form. All arguments may be present or absent.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_init
```

⟨*Quantum numbers: interfaces*⟩≡
```
    interface quantum_numbers_init
       module procedure quantum_numbers_init0_f
       module procedure quantum_numbers_init0_c
       module procedure quantum_numbers_init0_h
       module procedure quantum_numbers_init0_fc
       module procedure quantum_numbers_init0_fh
```

```
      module procedure quantum_numbers_init0_ch
      module procedure quantum_numbers_init0_fch
      module procedure quantum_numbers_init1_f
      module procedure quantum_numbers_init1_c
      module procedure quantum_numbers_init1_h
      module procedure quantum_numbers_init1_fc
      module procedure quantum_numbers_init1_fh
      module procedure quantum_numbers_init1_ch
      module procedure quantum_numbers_init1_fch
    end interface
```

⟨*Quantum numbers: procedures*⟩≡
```
    subroutine quantum_numbers_init0_f (qn, flv)
      type(quantum_numbers_t), intent(out) :: qn
      type(flavor_t), intent(in) :: flv
      qn%f = flv
    end subroutine quantum_numbers_init0_f

    subroutine quantum_numbers_init0_c (qn, col)
      type(quantum_numbers_t), intent(out) :: qn
      type(color_t), intent(in) :: col
      qn%c = col
    end subroutine quantum_numbers_init0_c

    subroutine quantum_numbers_init0_h (qn, hel)
      type(quantum_numbers_t), intent(out) :: qn
      type(helicity_t), intent(in) :: hel
      qn%h = hel
    end subroutine quantum_numbers_init0_h

    subroutine quantum_numbers_init0_fc (qn, flv, col)
      type(quantum_numbers_t), intent(out) :: qn
      type(flavor_t), intent(in) :: flv
      type(color_t), intent(in) :: col
      qn%f = flv
      qn%c = col
    end subroutine quantum_numbers_init0_fc

    subroutine quantum_numbers_init0_fh (qn, flv, hel)
      type(quantum_numbers_t), intent(out) :: qn
      type(flavor_t), intent(in) :: flv
      type(helicity_t), intent(in) :: hel
      qn%f = flv
      qn%h = hel
    end subroutine quantum_numbers_init0_fh

    subroutine quantum_numbers_init0_ch (qn, col, hel)
      type(quantum_numbers_t), intent(out) :: qn
      type(color_t), intent(in) :: col
      type(helicity_t), intent(in) :: hel
      qn%c = col
      qn%h = hel
    end subroutine quantum_numbers_init0_ch
```

```
subroutine quantum_numbers_init0_fch (qn, flv, col, hel)
  type(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
  type(helicity_t), intent(in) :: hel
  qn%f = flv
  qn%c = col
  qn%h = hel
end subroutine quantum_numbers_init0_fch

subroutine quantum_numbers_init1_f (qn, flv)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_f (qn(i), flv(i))
  end do
end subroutine quantum_numbers_init1_f

subroutine quantum_numbers_init1_c (qn, col)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(color_t), dimension(:), intent(in) :: col
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_c (qn(i), col(i))
  end do
end subroutine quantum_numbers_init1_c

subroutine quantum_numbers_init1_h (qn, hel)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(helicity_t), dimension(:), intent(in) :: hel
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_h (qn(i), hel(i))
  end do
end subroutine quantum_numbers_init1_h

subroutine quantum_numbers_init1_fc (qn, flv, col)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  type(color_t), dimension(:), intent(in) :: col
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_fc (qn(i), flv(i), col(i))
  end do
end subroutine quantum_numbers_init1_fc

subroutine quantum_numbers_init1_fh (qn, flv, hel)
  type(quantum_numbers_t), dimension(:), intent(out) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  type(helicity_t), dimension(:), intent(in) :: hel
  integer :: i
  do i = 1, size (qn)
     call quantum_numbers_init0_fh (qn(i), flv(i), hel(i))
```

```
      end do
    end subroutine quantum_numbers_init1_fh

    subroutine quantum_numbers_init1_ch (qn, col, hel)
      type(quantum_numbers_t), dimension(:), intent(out) :: qn
      type(color_t), dimension(:), intent(in) :: col
      type(helicity_t), dimension(:), intent(in) :: hel
      integer :: i
      do i = 1, size (qn)
        call quantum_numbers_init0_ch (qn(i), col(i), hel(i))
      end do
    end subroutine quantum_numbers_init1_ch

    subroutine quantum_numbers_init1_fch (qn, flv, col, hel)
      type(quantum_numbers_t), dimension(:), intent(out) :: qn
      type(flavor_t), dimension(:), intent(in) :: flv
      type(color_t), dimension(:), intent(in) :: col
      type(helicity_t), dimension(:), intent(in) :: hel
      integer :: i
      do i = 1, size (qn)
        call quantum_numbers_init0_fch (qn(i), flv(i), col(i), hel(i))
      end do
    end subroutine quantum_numbers_init1_fch
```

7.4.2 I/O

Write the quantum numbers in condensed form, enclosed by square brackets.
For convenience, introduce also an array version.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_write
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_write
      module procedure quantum_numbers_write_single
      module procedure quantum_numbers_write_array
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_write_single (qn, unit)
      type(quantum_numbers_t), intent(in) :: qn
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no")  "["
      if (flavor_is_defined (qn%f)) then
        call flavor_write (qn%f, u)
        if (color_is_defined (qn%c) .or. helicity_is_defined (qn%h)) &
             write (u, "(1x)", advance="no")
      end if
      if (color_is_defined (qn%c) .or. color_is_ghost (qn%c)) then
        call color_write (qn%c, u)
        if (helicity_is_defined (qn%h))  write (u, "(1x)", advance="no")
      end if
```

```
      if (helicity_is_defined (qn%h)) then
         call helicity_write (qn%h, u)
      end if
      write (u, "(A)", advance="no")  "]"
   end subroutine quantum_numbers_write_single

   subroutine quantum_numbers_write_array (qn, unit)
     type(quantum_numbers_t), dimension(:), intent(in) :: qn
     integer, intent(in), optional :: unit
     integer :: i
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     write (u, "(A)", advance="no")  "["
     do i = 1, size (qn)
        if (i > 1)  write (u, "(A)", advance="no")  " / "
        if (flavor_is_defined (qn(i)%f)) then
           call flavor_write (qn(i)%f, u)
           if (color_is_defined (qn(i)%c) .or. helicity_is_defined (qn(i)%h)) &
                 write (u, "(1x)", advance="no")
        end if
        if (color_is_defined (qn(i)%c) .or. color_is_ghost (qn(i)%c)) then
           call color_write (qn(i)%c, u)
           if (helicity_is_defined (qn(i)%h))  write (u, "(1x)", advance="no")
        end if
        if (helicity_is_defined (qn(i)%h)) then
           call helicity_write (qn(i)%h, u)
        end if
     end do
     write (u, "(A)", advance="no")  "]"
   end subroutine quantum_numbers_write_array
```

Binary I/O.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_write_raw
    public :: quantum_numbers_read_raw
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_write_raw (qn, u)
      type(quantum_numbers_t), intent(in) :: qn
      integer, intent(in) :: u
      call flavor_write_raw (qn%f, u)
      call color_write_raw (qn%c, u)
      call helicity_write_raw (qn%h, u)
    end subroutine quantum_numbers_write_raw

    subroutine quantum_numbers_read_raw (qn, u, iostat)
      type(quantum_numbers_t), intent(out) :: qn
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      call flavor_read_raw (qn%f, u, iostat=iostat)
      call color_read_raw (qn%c, u, iostat=iostat)
      call helicity_read_raw (qn%h, u, iostat=iostat)
    end subroutine quantum_numbers_read_raw
```

7.4.3 Accessing contents

Color and helicity can be done by elemental functions. Flavor needs explicit specifics because of the pointer assignment.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_get_flavor
    public :: quantum_numbers_get_color
    public :: quantum_numbers_get_helicity
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_get_flavor
       module procedure quantum_numbers_get_flavor0
       module procedure quantum_numbers_get_flavor1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function quantum_numbers_get_flavor0 (qn) result (flv)
      type(flavor_t) :: flv
      type(quantum_numbers_t), intent(in) :: qn
      flv = qn%f
    end function quantum_numbers_get_flavor0

    function quantum_numbers_get_flavor1 (qn) result (flv)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      type(flavor_t), dimension(size(qn)) :: flv
      integer :: i
      do i = 1, size (qn)
         flv(i) = qn(i)%f
      end do
    end function quantum_numbers_get_flavor1

    elemental function quantum_numbers_get_color (qn) result (col)
      type(color_t) :: col
      type(quantum_numbers_t), intent(in) :: qn
      col = qn%c
    end function quantum_numbers_get_color

    elemental function quantum_numbers_get_helicity (qn) result (hel)
      type(helicity_t) :: hel
      type(quantum_numbers_t), intent(in) :: qn
      hel = qn%h
    end function quantum_numbers_get_helicity
```

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_set_flavor
    public :: quantum_numbers_set_color
    public :: quantum_numbers_set_helicity
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_set_flavor
       module procedure quantum_numbers_set_flavor0
       module procedure quantum_numbers_set_flavor1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
subroutine quantum_numbers_set_flavor0 (qn, flv)
  type(quantum_numbers_t), intent(inout) :: qn
  type(flavor_t), intent(in) :: flv
  qn%f = flv
end subroutine quantum_numbers_set_flavor0

subroutine quantum_numbers_set_flavor1 (qn, flv)
  type(quantum_numbers_t), dimension(:), intent(inout) :: qn
  type(flavor_t), dimension(:), intent(in) :: flv
  integer :: i
  do i = 1, size (flv)
     qn(i)%f = flv(i)
  end do
end subroutine quantum_numbers_set_flavor1

elemental subroutine quantum_numbers_set_color (qn, col)
  type(quantum_numbers_t), intent(inout) :: qn
  type(color_t), intent(in) :: col
  qn%c = col
end subroutine quantum_numbers_set_color

elemental subroutine quantum_numbers_set_helicity (qn, hel)
  type(quantum_numbers_t), intent(inout) :: qn
  type(helicity_t), intent(in) :: hel
  qn%h = hel
end subroutine quantum_numbers_set_helicity
```

This just resets the ghost property of the color/helicity part:

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_set_color_ghost
public :: quantum_numbers_set_helicity_ghost
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental subroutine quantum_numbers_set_color_ghost (qn, ghost)
  type(quantum_numbers_t), intent(inout) :: qn
  logical, intent(in) :: ghost
  call color_set_ghost (qn%c, ghost)
end subroutine quantum_numbers_set_color_ghost

elemental subroutine quantum_numbers_set_helicity_ghost (qn, ghost)
  type(quantum_numbers_t), intent(inout) :: qn
  logical, intent(in) :: ghost
  call helicity_set_ghost (qn%h, ghost)
end subroutine quantum_numbers_set_helicity_ghost
```

Assign a model to the flavor part of quantum numbers.

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_set_model
```

⟨*Quantum numbers: interfaces*⟩+≡

```
interface quantum_numbers_set_model
   module procedure quantum_numbers_set_model_single
   module procedure quantum_numbers_set_model_array
```

```
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_set_model_single (qn, model)
      type(quantum_numbers_t), intent(inout) :: qn
      type(model_t), intent(in), target :: model
      call flavor_set_model (qn%f, model)
    end subroutine quantum_numbers_set_model_single

    subroutine quantum_numbers_set_model_array (qn, model)
      type(quantum_numbers_t), dimension(:), intent(inout) :: qn
      type(model_t), intent(in), target :: model
      call flavor_set_model (qn%f, model)
    end subroutine quantum_numbers_set_model_array
```

This is a convenience function: return the color type for the flavor (array).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_get_color_type
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_get_color_type (qn) result (color_type)
      integer :: color_type
      type(quantum_numbers_t), intent(in) :: qn
      color_type = flavor_get_color_type (qn%f)
    end function quantum_numbers_get_color_type
```

7.4.4 Predicates

Check if the flavor index is valid (including UNDEFINED).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_valid
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_valid (qn) result (valid)
      logical :: valid
      type(quantum_numbers_t), intent(in) :: qn
      valid = flavor_is_valid (qn%f)
    end function quantum_numbers_are_valid
```

Check if the flavor part has its particle-data pointer associated (debugging aid).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_associated
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_associated (qn) result (flag)
      logical :: flag
      type(quantum_numbers_t), intent(in) :: qn
      flag = flavor_is_associated (qn%f)
    end function quantum_numbers_are_associated
```

Check if the helicity and color quantum numbers are diagonal. (Unpolarized/colorless also counts as diagonal.) Flavor is diagonal by definition.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_diagonal
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_diagonal (qn) result (diagonal)
      logical :: diagonal
      type(quantum_numbers_t), intent(in) :: qn
      diagonal = helicity_is_diagonal (qn%h) .and. color_is_diagonal (qn%c)
    end function quantum_numbers_are_diagonal
```

Check if the color and/or helicity part has the ghost property.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_is_color_ghost
    public :: quantum_numbers_is_helicity_ghost
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_is_color_ghost (qn) result (ghost)
      logical :: ghost
      type(quantum_numbers_t), intent(in) :: qn
      ghost = color_is_ghost (qn%c)
    end function quantum_numbers_is_color_ghost

    elemental function quantum_numbers_is_helicity_ghost (qn) result (ghost)
      logical :: ghost
      type(quantum_numbers_t), intent(in) :: qn
      ghost = helicity_is_ghost (qn%h)
    end function quantum_numbers_is_helicity_ghost
```

7.4.5 Comparisons

Matching and equality is derived from the individual quantum numbers. The variant `fhmatch` matches only flavor and helicity. The variant `dhmatch` matches only diagonal helicity, if the matching helicity is undefined.

⟨*Quantum numbers: public*⟩+≡
```
    public :: operator(.match.)
    public :: operator(.fmatch.)
    public :: operator(.fhmatch.)
    public :: operator(.dhmatch.)
    public :: operator(==)
    public :: operator(/=)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface operator(.match.)
       module procedure quantum_numbers_match
    end interface
    interface operator(.fmatch.)
       module procedure quantum_numbers_match_f
    end interface
    interface operator(.fhmatch.)
       module procedure quantum_numbers_match_fh
    end interface
```

```
      interface operator(.dhmatch.)
         module procedure quantum_numbers_match_hel_diag
      end interface
      interface operator(==)
         module procedure quantum_numbers_eq
      end interface
      interface operator(/=)
         module procedure quantum_numbers_neq
      end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
      elemental function quantum_numbers_match (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f) .and. &
              (qn1%c .match. qn2%c) .and. &
              (qn1%h .match. qn2%h)
      end function quantum_numbers_match

      elemental function quantum_numbers_match_f (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f)
      end function quantum_numbers_match_f

      elemental function quantum_numbers_match_fh (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f) .and. &
              (qn1%h .match. qn2%h)
      end function quantum_numbers_match_fh

      elemental function quantum_numbers_match_hel_diag (qn1, qn2) result (match)
        logical :: match
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        match = (qn1%f .match. qn2%f) .and. &
              (qn1%c .match. qn2%c) .and. &
              (qn1%h .dmatch. qn2%h)
      end function quantum_numbers_match_hel_diag

      elemental function quantum_numbers_eq (qn1, qn2) result (eq)
        logical :: eq
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        eq = (qn1%f == qn2%f) .and. &
              (qn1%c == qn2%c) .and. &
              (qn1%h == qn2%h)
      end function quantum_numbers_eq

      elemental function quantum_numbers_neq (qn1, qn2) result (neq)
        logical :: neq
        type(quantum_numbers_t), intent(in) :: qn1, qn2
        neq = (qn1%f /= qn2%f) .or. &
              (qn1%c /= qn2%c) .or. &
              (qn1%h /= qn2%h)
      end function quantum_numbers_neq
```

Two sets of quantum numbers are compatible if the individual quantum numbers are compatible, depending on the mask. Flavor has to match, regardless of the flavor mask.

If the color flag is set, color is compatible if the ghost property is identical. If the color flag is unset, color has to be identical. I.e., if the flag is set, the color amplitudes can interfere. If it is not set, they must be identical, and there must be no ghost. The latter property is used for expanding physical color flows.

Helicity is compatible if the mask is unset, otherwise it has to match. This determines if two amplitudes can be multiplied (no mask) or traced (mask).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_compatible
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_compatible (qn1, qn2, mask) &
       result (flag)
      logical :: flag
      type(quantum_numbers_t), intent(in) :: qn1, qn2
      type(quantum_numbers_mask_t), intent(in) :: mask
      if (mask%h .or. mask%hd) then
         flag = (qn1%f .match. qn2%f) .and. (qn1%h .match. qn2%h)
      else
         flag = (qn1%f .match. qn2%f)
      end if
      if (mask%c) then
         flag = flag .and. (color_is_ghost (qn1%c) .eqv. color_is_ghost (qn2%c))
      else
         flag = flag .and. &
              .not. (color_is_ghost (qn1%c) .or. color_is_ghost (qn2%c)) .and. &
              (qn1%c == qn2%c)
      end if
    end function quantum_numbers_are_compatible
```

This is the analog for a single quantum-number set. We just check for color ghosts; they are excluded if the color mask is unset (color-flow expansion).

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_are_physical
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_are_physical (qn, mask) result (flag)
      logical :: flag
      type(quantum_numbers_t), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      if (mask%c) then
         flag = .true.
      else
         flag = .not. color_is_ghost (qn%c)
      end if
    end function quantum_numbers_are_physical
```

7.4.6 Operations

Inherited from the color component: reassign color indices in canonical order.

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_canonicalize_color
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_numbers_canonicalize_color (qn)
  type(quantum_numbers_t), dimension(:), intent(inout) :: qn
  call color_canonicalize (qn%c)
end subroutine quantum_numbers_canonicalize_color
```

Inherited from the color component: make a color map for two matching quantum-number arrays.

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_set_color_map
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_numbers_set_color_map (map, qn1, qn2)
  integer, dimension(:,:), intent(out), allocatable :: map
  type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
  call set_color_map (map, qn1%c, qn2%c)
end subroutine quantum_numbers_set_color_map
```

Inherited from the color component: translate the color part using a color-map array

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_translate_color
```

⟨*Quantum numbers: interfaces*⟩+≡
```
interface quantum_numbers_translate_color
   module procedure quantum_numbers_translate_color0
   module procedure quantum_numbers_translate_color1
end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_numbers_translate_color0 (qn, map, offset)
  type(quantum_numbers_t), intent(inout) :: qn
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in), optional :: offset
  call color_translate (qn%c, map, offset)
end subroutine quantum_numbers_translate_color0

subroutine quantum_numbers_translate_color1 (qn, map, offset)
  type(quantum_numbers_t), dimension(:), intent(inout) :: qn
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in), optional :: offset
  call color_translate (qn%c, map, offset)
end subroutine quantum_numbers_translate_color1
```

Inherited from the color component: return the color index with highest absolute value

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_get_max_color_value
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_get_max_color_value
       module procedure quantum_numbers_get_max_color_value0
       module procedure quantum_numbers_get_max_color_value1
       module procedure quantum_numbers_get_max_color_value2
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function quantum_numbers_get_max_color_value0 (qn) result (cmax)
      integer :: cmax
      type(quantum_numbers_t), intent(in) :: qn
      cmax = color_get_max_value (qn%c)
    end function quantum_numbers_get_max_color_value0

    function quantum_numbers_get_max_color_value1 (qn) result (cmax)
      integer :: cmax
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      cmax = color_get_max_value (qn%c)
    end function quantum_numbers_get_max_color_value1

    function quantum_numbers_get_max_color_value2 (qn) result (cmax)
      integer :: cmax
      type(quantum_numbers_t), dimension(:,:), intent(in) :: qn
      cmax = color_get_max_value (qn%c)
    end function quantum_numbers_get_max_color_value2
```

Inherited from the color component: add an offset to the indices of the color
part

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_add_color_offset
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_add_color_offset (qn, offset)
      type(quantum_numbers_t), intent(inout) :: qn
      integer, intent(in) :: offset
      call color_add_offset (qn%c, offset)
    end subroutine quantum_numbers_add_color_offset
```

Given a quantum number array, return all possible color contractions, leaving
the other quantum numbers intact.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_number_array_make_color_contractions
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_number_array_make_color_contractions (qn_in, qn_out)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
      type(quantum_numbers_t), dimension(:,:), intent(out), allocatable :: qn_out
      type(color_t), dimension(:,:), allocatable :: col
      integer :: i
      call color_array_make_contractions (qn_in%c, col)
      allocate (qn_out (size (col, 1), size (col, 2)))
      do i = 1, size (qn_out, 2)
         qn_out(:,i)%f = qn_in%f
```

```
        qn_out(:,i)%c = col(:,i)
        qn_out(:,i)%h = qn_in%h
      end do
    end subroutine quantum_number_array_make_color_contractions
```

Inherited from the color component: invert the color, switching particle/antiparticle.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_invert_color
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_invert_color (qn)
      type(quantum_numbers_t), intent(inout) :: qn
      call color_invert (qn%c)
    end subroutine quantum_numbers_invert_color
```

Merge two quantum number sets: for each entry, if both are defined, combine them to an off-diagonal entry (meaningful only if the input was diagonal). If either entry is undefined, take the defined one.

For flavor, off-diagonal entries are invalid, so both flavors must be equal, otherwise an invalid flavor is inserted.

⟨*Quantum numbers: public*⟩+≡
```
    public :: operator(.merge.)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface operator(.merge.)
       module procedure merge_quantum_numbers0
       module procedure merge_quantum_numbers1
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function merge_quantum_numbers0 (qn1, qn2) result (qn3)
      type(quantum_numbers_t) :: qn3
      type(quantum_numbers_t), intent(in) :: qn1, qn2
      qn3%f = qn1%f .merge. qn2%f
      qn3%c = qn1%c .merge. qn2%c
      qn3%h = qn1%h .merge. qn2%h
    end function merge_quantum_numbers0

    function merge_quantum_numbers1 (qn1, qn2) result (qn3)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
      type(quantum_numbers_t), dimension(size(qn1)) :: qn3
      qn3%f = qn1%f .merge. qn2%f
      qn3%c = qn1%c .merge. qn2%c
      qn3%h = qn1%h .merge. qn2%h
    end function merge_quantum_numbers1
```

7.4.7 The quantum number mask

The quantum numbers mask is true for quantum numbers that should be ignored or summed over. The three mandatory entries correspond to flavor, color, and helicity, respectively.

743

There is an additional entry `cg`: If false, the color-ghosts property should be kept even if color is ignored. This is relevant only if `c` is set, otherwise it is always false.

The flag `hd` tells that only diagonal entries in helicity should be kept. If `h` is set, `hd` is irrelevant and will be kept `.false.`

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_mask_t
```

⟨*Quantum numbers: types*⟩+≡
```
type :: quantum_numbers_mask_t
  private
  logical :: f = .false.
  logical :: c = .false.
  logical :: cg = .false.
  logical :: h = .false.
  logical :: hd = .false.
end type quantum_numbers_mask_t
```

Define a quantum number mask: Constructor form

⟨*Quantum numbers: public*⟩+≡
```
public :: new_quantum_numbers_mask
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function new_quantum_numbers_mask &
    (mask_f, mask_c, mask_h, mask_cg, mask_hd) result (mask)
  type(quantum_numbers_mask_t) :: mask
  logical, intent(in) :: mask_f, mask_c, mask_h
  logical, intent(in), optional :: mask_cg
  logical, intent(in), optional :: mask_hd
  call quantum_numbers_mask_init &
      (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
end function new_quantum_numbers_mask
```

Define quantum numbers: Initializer form

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_mask_init
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental subroutine quantum_numbers_mask_init &
    (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
  type(quantum_numbers_mask_t), intent(out) :: mask
  logical, intent(in) :: mask_f, mask_c, mask_h
  logical, intent(in), optional :: mask_cg, mask_hd
  mask%f = mask_f
  mask%c = mask_c
  mask%h = mask_h
  if (present (mask_cg)) then
     if (mask%c)  mask%cg = mask_cg
  else
     mask%cg = mask_c
  end if
  if (present (mask_hd)) then
     if (.not. mask%h)  mask%hd = mask_hd
  end if
```

```
      end subroutine quantum_numbers_mask_init
```

Write a quantum numbers mask

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_write
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_mask_write
       module procedure quantum_numbers_mask_write_single
       module procedure quantum_numbers_mask_write_array
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    subroutine quantum_numbers_mask_write_single (mask, unit)
      type(quantum_numbers_mask_t), intent(in) :: mask
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no") "["
      write (u, "(L1)", advance="no")  mask%f
      write (u, "(L1)", advance="no")  mask%c
      if (.not.mask%cg)  write (u, "('g')", advance="no")
      write (u, "(L1)", advance="no")  mask%h
      if (mask%hd)  write (u, "('d')", advance="no")
      write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_mask_write_single

    subroutine quantum_numbers_mask_write_array (mask, unit)
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)", advance="no") "["
      do i = 1, size (mask)
         if (i > 1)  write (u, "(A)", advance="no")  "/"
         write (u, "(L1)", advance="no")  mask(i)%f
         write (u, "(L1)", advance="no")  mask(i)%c
         if (.not.mask(i)%cg)  write (u, "('g')", advance="no")
         write (u, "(L1)", advance="no")  mask(i)%h
         if (mask(i)%hd)  write (u, "('d')", advance="no")
      end do
      write (u, "(A)", advance="no") "]"
    end subroutine quantum_numbers_mask_write_array
```

7.4.8   Setting mask components

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_set_flavor
    public :: quantum_numbers_mask_set_color
    public :: quantum_numbers_mask_set_helicity
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_mask_set_flavor (mask, mask_f)
      type(quantum_numbers_mask_t), intent(inout) :: mask
```

```
    logical, intent(in) :: mask_f
    mask%f = mask_f
  end subroutine quantum_numbers_mask_set_flavor

  elemental subroutine quantum_numbers_mask_set_color (mask, mask_c, mask_cg)
    type(quantum_numbers_mask_t), intent(inout) :: mask
    logical, intent(in) :: mask_c
    logical, intent(in), optional :: mask_cg
    mask%c = mask_c
    if (present (mask_cg)) then
       if (mask%c)  mask%cg = mask_cg
    else
       mask%cg = mask_c
    end if
  end subroutine quantum_numbers_mask_set_color

  elemental subroutine quantum_numbers_mask_set_helicity (mask, mask_h, mask_hd)
    type(quantum_numbers_mask_t), intent(inout) :: mask
    logical, intent(in) :: mask_h
    logical, intent(in), optional :: mask_hd
    mask%h = mask_h
    if (present (mask_hd)) then
       if (.not. mask%h)  mask%hd = mask_hd
    end if
  end subroutine quantum_numbers_mask_set_helicity
```

The following routines assign part of a mask, depending on the flags given.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_assign
```

⟨*Quantum numbers: procedures*⟩+≡
```
  elemental subroutine quantum_numbers_mask_assign &
       (mask, mask_in, flavor, color, helicity)
    type(quantum_numbers_mask_t), intent(inout) :: mask
    type(quantum_numbers_mask_t), intent(in) :: mask_in
    logical, intent(in), optional :: flavor, color, helicity
    if (present (flavor)) then
       if (flavor) then
          mask%f = mask_in%f
       end if
    end if
    if (present (color)) then
       if (color) then
          mask%c = mask_in%c
          mask%cg = mask_in%cg
       end if
    end if
    if (present (helicity)) then
       if (helicity) then
          mask%h = mask_in%h
          mask%hd = mask_in%hd
       end if
    end if
  end subroutine quantum_numbers_mask_assign
```

746

7.4.9 Mask predicates

Return true if either one of the entries is set:

⟨*Quantum numbers: public*⟩+≡
```
    public :: any
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface any
       module procedure quantum_numbers_mask_any
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    function quantum_numbers_mask_any (mask) result (match)
      logical :: match
      type(quantum_numbers_mask_t), intent(in) :: mask
      match = mask%f .or. mask%c .or. mask%h .or. mask%hd
    end function quantum_numbers_mask_any
```

7.4.10 Operators

The OR operation is applied to all components.

⟨*Quantum numbers: public*⟩+≡
```
    public :: operator(.or.)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface operator(.or.)
       module procedure quantum_numbers_mask_or
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_mask_or (mask1, mask2) result (mask)
       type(quantum_numbers_mask_t) :: mask
       type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
       mask%f = mask1%f .or. mask2%f
       mask%c = mask1%c .or. mask2%c
       if (mask%c)  mask%cg = mask1%cg .or. mask2%cg
       mask%h = mask1%h .or. mask2%h
       if (.not. mask%h)  mask%hd = mask1%hd .or. mask2%hd
     end function quantum_numbers_mask_or
```

7.4.11 Mask comparisons

Return true if the two masks are equivalent / differ:

⟨*Quantum numbers: public*⟩+≡
```
    public :: operator(.eqv.)
    public :: operator(.neqv.)
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface operator(.eqv.)
       module procedure quantum_numbers_mask_eqv
    end interface
    interface operator(.neqv.)
       module procedure quantum_numbers_mask_neqv
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_mask_eqv (mask1, mask2) result (eqv)
      logical :: eqv
      type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
      eqv = (mask1%f .eqv. mask2%f) .and. &
            (mask1%c .eqv. mask2%c) .and. &
            (mask1%cg .eqv. mask2%cg) .and. &
            (mask1%h .eqv. mask2%h) .and. &
            (mask1%hd .eqv. mask2%hd)
    end function quantum_numbers_mask_eqv

    elemental function quantum_numbers_mask_neqv (mask1, mask2) result (neqv)
      logical :: neqv
      type(quantum_numbers_mask_t), intent(in) :: mask1, mask2
      neqv = (mask1%f .neqv. mask2%f) .or. &
            (mask1%c .neqv. mask2%c) .or. &
            (mask1%cg .neqv. mask2%cg) .or. &
            (mask1%h .neqv. mask2%h) .or. &
            (mask1%hd .neqv. mask2%hd)
    end function quantum_numbers_mask_neqv
```

7.4.12 Apply a mask

Applying a mask to the quantum number object means undefining those entries
where the mask is set. The others remain unaffected.

The `hd` mask has the special property that it "diagonalizes" helicity, i.e., the
second helicity entry is dropped and the result is a diagonal helicity quantum
number.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_undefine
    public :: quantum_numbers_undefined
```

⟨*Quantum numbers: interfaces*⟩+≡
```
    interface quantum_numbers_undefined
       module procedure quantum_numbers_undefined0
       module procedure quantum_numbers_undefined1
       module procedure quantum_numbers_undefined11
    end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental subroutine quantum_numbers_undefine (qn, mask)
      type(quantum_numbers_t), intent(inout) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      if (mask%f)  call flavor_undefine (qn%f)
      if (mask%c)  call color_undefine (qn%c, undefine_ghost=mask%cg)
```

```
      if (mask%h) then
         call helicity_undefine (qn%h)
      else if (mask%hd) then
         if (.not. helicity_is_diagonal (qn%h)) then
            call helicity_diagonalize (qn%h)
         end if
      end if
   end subroutine quantum_numbers_undefine

   function quantum_numbers_undefined0 (qn, mask) result (qn_new)
      type(quantum_numbers_t), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      type(quantum_numbers_t) :: qn_new
      qn_new = qn
      call quantum_numbers_undefine (qn_new, mask)
   end function quantum_numbers_undefined0

   function quantum_numbers_undefined1 (qn, mask) result (qn_new)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      type(quantum_numbers_t), dimension(size(qn)) :: qn_new
      qn_new = qn
      call quantum_numbers_undefine (qn_new, mask)
   end function quantum_numbers_undefined1

   function quantum_numbers_undefined11 (qn, mask) result (qn_new)
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      type(quantum_numbers_t), dimension(size(qn)) :: qn_new
      qn_new = qn
      call quantum_numbers_undefine (qn_new, mask)
   end function quantum_numbers_undefined11
```

Return true if the input quantum number set has entries that would be removed by the applied mask, e.g., if polarization is defined but `mask%h` is set:

⟨*Quantum numbers: public*⟩+≡
```
   public :: quantum_numbers_are_redundant
```

⟨*Quantum numbers: procedures*⟩+≡
```
   elemental function quantum_numbers_are_redundant (qn, mask) &
        result (redundant)
      logical :: redundant
      type(quantum_numbers_t), intent(in) :: qn
      type(quantum_numbers_mask_t), intent(in) :: mask
      redundant = .false.
      if (mask%f) then
         redundant = flavor_is_defined (qn%f)
      end if
      if (mask%c) then
         redundant = color_is_defined (qn%c)
      end if
      if (mask%h) then
         redundant = helicity_is_defined (qn%h)
      else if (mask%hd) then
```

```
        redundant = .not. helicity_is_diagonal (qn%h)
      end if
    end function quantum_numbers_are_redundant
```

Return true if the helicity flag is set or the diagonal-helicity flag is set.

⟨*Quantum numbers: public*⟩+≡
```
    public :: quantum_numbers_mask_diagonal_helicity
```

⟨*Quantum numbers: procedures*⟩+≡
```
    elemental function quantum_numbers_mask_diagonal_helicity (mask) &
        result (flag)
      logical :: flag
      type(quantum_numbers_mask_t), intent(in) :: mask
      flag = mask%h .or. mask%hd
    end function quantum_numbers_mask_diagonal_helicity
```

7.5 State matrices

This module deals with the internal state of a particle system, i.e., with its density matrix in flavor, color, and helicity space.

⟨state_matrices.f90⟩≡
  ⟨*File header*⟩

  ```
  module state_matrices
  ```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
  ```
    use limits, only: FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
  ```

  ⟨*Standard module head*⟩

  ⟨*State matrices: public*⟩

  ⟨*State matrices: parameters*⟩

  ⟨*State matrices: types*⟩

  ⟨*State matrices: interfaces*⟩

  ```
  contains
  ```

  ⟨*State matrices: procedures*⟩

  ⟨*State matrices: tests*⟩

  ```
  end module state_matrices
  ```

7.5.1 Nodes of the quantum state trie

A quantum state object represents an unnormalized density matrix, i.e., an array of possibilities for flavor, color, and helicity indices with associated complex values. Physically, the trace of this matrix is the summed squared matrix element for an interaction, and the matrix elements divided by this value correspond to the flavor-color-helicity density matrix. (Flavor and color are diagonal.)

We store density matrices as tries, that is, as trees where each branching represents the possible quantum numbers of a particle. The first branching is the first particle in the system. A leaf (the node corresponding to the last particle) contains the value of the matrix element.

Each node contains a flavor, color, and helicity entry. Note that each of those entries may be actually undefined, so we can also represent, e.g., unpolarized particles.

The value is meaningful only for leaves, which have no child nodes. There is a pointer to the parent node which allows for following the trie downwards from a leaf, it is null for a root node. The child nodes are implemented as a list, so there is a pointer to the first and last child, and each node also has a `next` pointer to the next sibling.

The root node does not correspond to a particle, only its children do. The quantum numbers of the root node are irrelevant and will not be set. However, we use a common type for the three classes (root, branch, leaf); they may easily be distinguished by the association status of parent and child.

Node type

The node is linked in all directions: the parent, the first and last in the list of children, and the previous and next sibling. This allows us for adding and removing nodes and whole branches anywhere in the trie. (Circular links are not allowed, however.). The node holds its associated set of quantum numbers. The integer index, which is set only for leaf nodes, is the index of the corresponding matrix element value within the state matrix.

Temporarily, matrix-element values may be stored within a leaf node. This is used during state-matrix factorization. When the state matrix is `freeze`d, these values are transferred to the matrix-element array within the host state matrix.

⟨*State matrices: types*⟩≡

```
type :: node_t
   private
   type(quantum_numbers_t) :: qn
   type(node_t), pointer :: parent => null ()
   type(node_t), pointer :: child_first => null ()
   type(node_t), pointer :: child_last => null ()
   type(node_t), pointer :: next => null ()
   type(node_t), pointer :: previous => null ()
   integer :: me_index = 0
   integer, dimension(:), allocatable :: me_count
   complex(default) :: me = 0
end type node_t
```

Operations on nodes

Recursively deallocate all children of the current node. This includes any values associated with the children.

⟨*State matrices: procedures*⟩≡

```
pure recursive subroutine node_delete_offspring (node)
   type(node_t), pointer :: node
   type(node_t), pointer :: child
   child => node%child_first
   do while (associated (child))
      node%child_first => node%child_first%next
      call node_delete_offspring (child)
      deallocate (child)
      child => node%child_first
   end do
```

```
      node%child_last => null ()
   end subroutine node_delete_offspring
```

Remove a node including its offspring. Adjust the pointers of parent and siblings, if necessary.

⟨*State matrices: procedures*⟩+≡
```
   pure subroutine node_delete (node)
     type(node_t), pointer :: node
     call node_delete_offspring (node)
     if (associated (node%previous)) then
        node%previous%next => node%next
     else if (associated (node%parent)) then
        node%parent%child_first => node%next
     end if
     if (associated (node%next)) then
        node%next%previous => node%previous
     else if (associated (node%parent)) then
        node%parent%child_last => node%previous
     end if
     deallocate (node)
   end subroutine node_delete
```

Append a child node

⟨*State matrices: procedures*⟩+≡
```
   subroutine node_append_child (node, child)
     type(node_t), target, intent(inout) :: node
     type(node_t), pointer :: child
     allocate (child)
     if (associated (node%child_last)) then
        node%child_last%next => child
        child%previous => node%child_last
     else
        node%child_first => child
     end if
     node%child_last => child
     child%parent => node
   end subroutine node_append_child
```

I/O

Output of a single node, no recursion. We print the quantum numbers in square brackets, then the value (if any).

⟨*State matrices: procedures*⟩+≡
```
   subroutine node_write (node, me_array, verbose, unit, testflag)
     type(node_t), intent(in) :: node
     complex(default), dimension(:), intent(in), optional :: me_array
     logical, intent(in), optional :: verbose, testflag
     integer, intent(in), optional :: unit
     logical :: verb
     integer :: u
     character(len=7) :: fmt
```

```
        call pac_fmt (fmt, FMT_19, FMT_17, testflag)
        verb = .false.;  if (present (verbose)) verb = verbose
        u = output_unit (unit);  if (u < 0)  return
        call quantum_numbers_write (node%qn, u)
        if (node%me_index /= 0) then
            write (u, "(A,I0,A)", advance="no")  " => ME(", node%me_index, ")"
            if (present (me_array)) then
                write (u, "(A)", advance="no")  " = "
                write (u, "('(',"  // fmt // "',',"  // fmt // "',')')", &
                    advance="no") pacify_complex (me_array(node%me_index))
            end if
        end if
        write (u, *)
        if (verb) then
            call ptr_write ("parent     ", node%parent)
            call ptr_write ("child_first", node%child_first)
            call ptr_write ("child_last ", node%child_last)
            call ptr_write ("next       ", node%next)
            call ptr_write ("previous   ", node%previous)
        end if
    contains
      subroutine ptr_write (label, node)
        character(*), intent(in) :: label
        type(node_t), pointer :: node
        if (associated (node)) then
            write (u, "(10x,A,1x,'->',1x)", advance="no") label
            call quantum_numbers_write (node%qn, u)
            write (u, *)
        end if
      end subroutine ptr_write
    end subroutine node_write
```

Recursive output of a node:

⟨*State matrices: procedures*⟩+≡

```
    recursive subroutine node_write_rec (node, me_array, verbose, &
          indent, unit, testflag)
      type(node_t), intent(in), target :: node
      complex(default), dimension(:), intent(in), optional :: me_array
      logical, intent(in), optional :: verbose, testflag
      integer, intent(in), optional :: indent
      integer, intent(in), optional :: unit
      type(node_t), pointer :: current
      logical :: verb
      integer :: i, u
      verb = .false.;  if (present (verbose))  verb = verbose
      i = 0;  if (present (indent)) i = indent
      u = output_unit (unit);  if (u < 0)  return
      current => node%child_first
      do while (associated (current))
          write (u, "(A)", advance="no")  repeat (" ", i)
          call node_write (current, me_array, verbose=verb, &
            unit=u, testflag=testflag)
          call node_write_rec (current, me_array, verbose=verb, &
            indent=i+2, unit=u, testflag = testflag)
```

```
          current => current%next
       end do
    end subroutine node_write_rec
```

Binary I/O. Matrix elements are written only for leaf nodes.

⟨*State matrices: procedures*⟩+≡

```
    recursive subroutine node_write_raw_rec (node, u)
      type(node_t), intent(in), target :: node
      integer, intent(in) :: u
      logical :: associated_child_first, associated_next
      call quantum_numbers_write_raw (node%qn, u)
      associated_child_first = associated (node%child_first)
      write (u) associated_child_first
      associated_next = associated (node%next)
      write (u) associated_next
      if (associated_child_first) then
         call node_write_raw_rec (node%child_first, u)
      else
         write (u)  node%me_index
         write (u)  node%me
      end if
      if (associated_next) then
         call node_write_raw_rec (node%next, u)
      end if
    end subroutine node_write_raw_rec

    recursive subroutine node_read_raw_rec (node, u, parent, iostat)
      type(node_t), intent(out), target :: node
      integer, intent(in) :: u
      type(node_t), intent(in), optional, target :: parent
      integer, intent(out), optional :: iostat
      logical :: associated_child_first, associated_next
      type(node_t), pointer :: child
      call quantum_numbers_read_raw (node%qn, u, iostat=iostat)
      read (u, iostat=iostat) associated_child_first
      read (u, iostat=iostat) associated_next
      if (present (parent))  node%parent => parent
      if (associated_child_first) then
         allocate (child)
         node%child_first => child
         node%child_last => null ()
         call node_read_raw_rec (child, u, node, iostat=iostat)
         do while (associated (child))
            child%previous => node%child_last
            node%child_last => child
            child => child%next
         end do
      else
         read (u, iostat=iostat)  node%me_index
         read (u, iostat=iostat)  node%me
      end if
      if (associated_next) then
         allocate (node%next)
         call node_read_raw_rec (node%next, u, parent, iostat=iostat)
```

```
      end if
    end subroutine node_read_raw_rec
```

7.5.2 State matrix

Definition

The quantum state object is a container that keeps and hides the root node.
For direct accessibility of values, they are stored in a separate array. The leaf
nodes of the quantum-number tree point to those values, once the state matrix
is finalized.

The `norm` component is redefined if a common factor is extracted from all
nodes.

⟨*State matrices: public*⟩≡
```
    public :: state_matrix_t
```

⟨*State matrices: types*⟩+≡
```
    type :: state_matrix_t
      private
      type(node_t), pointer :: root => null ()
      integer :: depth = 0
      integer :: n_matrix_elements = 0
      logical :: leaf_nodes_store_values = .false.
      integer :: n_counters = 0
      complex(default), dimension(:), allocatable :: me
      real(default) :: norm = 1
    end type state_matrix_t
```

This initializer allocates the root node but does not fill anything. We declare
whether values are stored within the nodes during state-matrix construction,
and how many counters should be maintained (default: none).

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_init
```

⟨*State matrices: procedures*⟩+≡
```
    elemental subroutine state_matrix_init (state, store_values, n_counters)
      type(state_matrix_t), intent(out) :: state
      logical, intent(in), optional :: store_values
      integer, intent(in), optional :: n_counters
      allocate (state%root)
      if (present (store_values)) &
         state%leaf_nodes_store_values = store_values
      if (present (n_counters)) state%n_counters = n_counters
    end subroutine state_matrix_init
```

This recursively deletes all children of the root node, restoring the initial state.
The matrix element array is not finalized, since it does not contain physical
entries, just pointers.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_final
```

⟨*State matrices: procedures*⟩+≡
```
    elemental subroutine state_matrix_final (state)
      type(state_matrix_t), intent(inout) :: state
      if (allocated (state%me))  deallocate (state%me)
      if (associated (state%root))  call node_delete (state%root)
      state%depth = 0
      state%n_matrix_elements = 0
    end subroutine state_matrix_final
```

Output: Present the tree as a nested list with appropriate indentation.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_write
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_write (state, unit, write_value_list, &
         verbose, testflag)
      type(state_matrix_t), intent(in) :: state
      logical, intent(in), optional :: write_value_list, verbose, testflag
      integer, intent(in), optional :: unit
      complex(default) :: me_dum
      character(len=7) :: fmt
      integer :: u
      integer :: i
      call pac_fmt (fmt, FMT_19, FMT_17, testflag)
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A," // fmt // ")") "State matrix:  norm = ", state%norm
      if (associated (state%root)) then
         if (allocated (state%me)) then
            call node_write_rec (state%root, state%me, verbose=verbose, &
               indent=1, unit=u, testflag=testflag)
         else
            call node_write_rec (state%root, verbose=verbose, indent=1, &
               unit=u, testflag=testflag)
         end if
      end if
      if (present (write_value_list)) then
         if (write_value_list .and. allocated (state%me)) then
            do i = 1, size (state%me)
               write (u, "(1x,I0,A)", advance="no")  i, ":"
               me_dum = state%me(i)
               if (real(state%me(i)) == -real(state%me(i))) then
                  me_dum = &
                       cmplx (0._default, aimag(me_dum), kind=default)
               end if
               if (aimag(me_dum) == -aimag(me_dum)) then
                  me_dum = &
                       cmplx (real(me_dum), 0._default, kind=default)
               end if
               write (u, "('('," // fmt // ",',','," // fmt // &
                  ",')')")  me_dum
            end do
         end if
      end if
    end subroutine state_matrix_write
```

Binary I/O. The auxiliary matrix-element array is not written, but reconstructed after reading the tree.

Note: To be checked. Might be broken, don't use (unless trivial).

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_write_raw
    public :: state_matrix_read_raw
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_write_raw (state, u)
      type(state_matrix_t), intent(in) :: state
      integer, intent(in) :: u
      logical :: associated_root
      associated_root = associated (state%root)
      write (u) associated_root
      if (associated_root) then
         write (u) state%depth
         write (u) state%norm
         call node_write_raw_rec (state%root, u)
      end if
    end subroutine state_matrix_write_raw

    subroutine state_matrix_read_raw (state, u, iostat)
      type(state_matrix_t), intent(out) :: state
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      logical :: associated_root
      read (u, iostat=iostat) associated_root
      if (associated_root) then
         read (u, iostat=iostat) state%depth
         read (u, iostat=iostat) state%norm
         call state_matrix_init (state)
         call node_read_raw_rec (state%root, u, iostat=iostat)
         call state_matrix_freeze (state)
      end if
    end subroutine state_matrix_read_raw
```

Properties of the quantum state

A state is defined if its root is allocated:

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_is_defined
```

⟨*State matrices: procedures*⟩+≡
```
    elemental function state_matrix_is_defined (state) result (defined)
      logical :: defined
      type(state_matrix_t), intent(in) :: state
      defined = associated (state%root)
    end function state_matrix_is_defined
```

A state is empty if its depth is zero:

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_is_empty
```

⟨*State matrices: procedures*⟩+≡
```
    elemental function state_matrix_is_empty (state) result (flag)
      logical :: flag
      type(state_matrix_t), intent(in) :: state
      flag = state%depth == 0
    end function state_matrix_is_empty
```

Return the number of matrix-element values.
⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_n_matrix_elements
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_n_matrix_elements (state) result (n)
      integer :: n
      type(state_matrix_t), intent(in) :: state
      n = state%n_matrix_elements
    end function state_matrix_get_n_matrix_elements
```

Return the number of leaves. This can be larger than the number of independent matrix elements.
⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_n_leaves
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_n_leaves (state) result (n)
      integer :: n
      type(state_matrix_t), intent(in) :: state
      type(state_iterator_t) :: it
      n = 0
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         n = n + 1
         call state_iterator_advance (it)
      end do
    end function state_matrix_get_n_leaves
```

Return the depth:
⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_depth
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_depth (state) result (depth)
      integer :: depth
      type(state_matrix_t), intent(in) :: state
      depth = state%depth
    end function state_matrix_get_depth
```

Return the norm:
⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_norm
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_norm (state) result (norm)
      real(default) :: norm
      type(state_matrix_t), intent(in) :: state
      norm = state%norm
    end function state_matrix_get_norm
```

Retrieving contents

Return the quantum number array, using an index. We have to scan the state
matrix since there is no shortcut.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_quantum_numbers
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_quantum_numbers (state, i) result (qn)
      type(state_matrix_t), intent(in), target :: state
      integer, intent(in) :: i
      type(quantum_numbers_t), dimension(state%depth) :: qn
      type(state_iterator_t) :: it
      integer :: k
      k = 0
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         k = k + 1
         if (k == i) then
            qn = state_iterator_get_quantum_numbers (it)
            return
         end if
         call state_iterator_advance (it)
      end do
    end function state_matrix_get_quantum_numbers
```

Return a single matrix element using its index. Works only if the shortcut array
is allocated.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_get_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_get_matrix_element (state, i) result (me)
      complex(default) :: me
      type(state_matrix_t), intent(in) :: state
      integer, intent(in) :: i
      if (allocated (state%me)) then
         me = state%me(i)
      else
         me = 0
      end if
    end function state_matrix_get_matrix_element
```

Return the color index with maximum absolute value that is present within the state matrix.

⟨*State matrices: public*⟩+≡
```
public :: state_matrix_get_max_color_value
```

⟨*State matrices: procedures*⟩+≡
```
function state_matrix_get_max_color_value (state) result (cmax)
  integer :: cmax
  type(state_matrix_t), intent(in) :: state
  if (associated (state%root)) then
     cmax = node_get_max_color_value (state%root)
  else
     cmax = 0
  end if
contains
  recursive function node_get_max_color_value (node) result (cmax)
    integer :: cmax
    type(node_t), intent(in), target :: node
    type(node_t), pointer :: current
    cmax = quantum_numbers_get_max_color_value (node%qn)
    current => node%child_first
    do while (associated (current))
       cmax = max (cmax, node_get_max_color_value (current))
       current => current%next
    end do
  end function node_get_max_color_value
end function state_matrix_get_max_color_value
```

Building the quantum state

The procedure generates a branch associated to the input array of quantum numbers. If the branch exists already, it is used.

Optionally, we set the matrix-element index, a value (which may be added to the previous one), and increment one of the possible counters. We may also return the matrix element index of the current node.

⟨*State matrices: public*⟩+≡
```
public :: state_matrix_add_state
```

⟨*State matrices: procedures*⟩+≡
```
subroutine state_matrix_add_state &
     (state, qn, index, value, sum_values, counter_index, me_index)
  type(state_matrix_t), intent(inout) :: state
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  integer, intent(in), optional :: index
  complex(default), intent(in), optional :: value
  logical, intent(in), optional :: sum_values
  integer, intent(in), optional :: counter_index
  integer, intent(out), optional :: me_index
  logical :: set_index, get_index, add
  set_index = present (index)
  get_index = present (me_index)
  add = .false.;  if (present (sum_values))  add = sum_values
  if (state%depth == 0) then
```

```fortran
        state%depth = size (qn)
     else if (state%depth /= size (qn)) then
        call state_matrix_write (state)
        call msg_bug ("State matrix: depth mismatch")
     end if
     if (size (qn) > 0)  call node_make_branch (state%root, qn)
  contains
     recursive subroutine node_make_branch (parent, qn)
        type(node_t), pointer :: parent
        type(quantum_numbers_t), dimension(:), intent(in) :: qn
        type(node_t), pointer :: child
        logical :: match
        match = .false.
        child => parent%child_first
        SCAN_CHILDREN: do while (associated (child))
           match = child%qn == qn(1)
           if (match)  exit SCAN_CHILDREN
           child => child%next
        end do SCAN_CHILDREN
        if (.not. match) then
           call node_append_child (parent, child)
           child%qn = qn(1)
        end if
        select case (size (qn))
        case (1)
           if (.not. match) then
              state%n_matrix_elements = state%n_matrix_elements + 1
              child%me_index = state%n_matrix_elements
           end if
           if (set_index) then
              child%me_index = index
           end if
           if (get_index) then
              me_index = child%me_index
           end if
           if (present (counter_index)) then
              if (.not. allocated (child%me_count)) then
                 allocate (child%me_count (state%n_counters))
                 child%me_count = 0
              end if
              child%me_count(counter_index) = child%me_count(counter_index) + 1
           end if
           if (present (value)) then
              if (add) then
                 child%me = child%me + value
              else
                 child%me = value
              end if
           end if
        case (2:)
           call node_make_branch (child, qn(2:))
        end select
     end subroutine node_make_branch
  end subroutine state_matrix_add_state
```

Remove irrelevant flavor/color/helicity labels and the corresponding branchings.
The masks indicate which particles are affected; the masks length should co-
incide with the depth of the trie (without the root node). Recursively scan
the whole tree, starting from the leaf nodes and working up to the root node.
If a mask entry is set for the current tree level, scan the children there. For
each child within that level make a new empty branch where the masked quan-
tum number is undefined. Then recursively combine all following children with
matching quantum number into this new node and move on.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_collapse
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_collapse (state, mask)
      type(state_matrix_t), intent(inout) :: state
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      type(state_matrix_t) :: red_state
      if (state_matrix_is_defined (state)) then
         call state_matrix_reduce (state, mask, red_state)
         call state_matrix_final (state)
         state = red_state
      end if
    end subroutine state_matrix_collapse
```

Transform the given state matrix into a reduced state matrix where some quan-
tum numbers are removed, as indicated by the mask. The procedure creates a
new state matrix, so the old one can be deleted after this if it is no longer used.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_reduce
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_reduce (state, mask, red_state)
      type(state_matrix_t), intent(in), target :: state
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      type(state_matrix_t), intent(out) :: red_state
      type(state_iterator_t) :: it
      type(quantum_numbers_t), dimension(size(mask)) :: qn
      call state_matrix_init (red_state)
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         qn = state_iterator_get_quantum_numbers (it)
         call quantum_numbers_undefine (qn, mask)
         call state_matrix_add_state (red_state, qn)
         call state_iterator_advance (it)
      end do
    end subroutine state_matrix_reduce
```

This subroutine sets up the matrix-element array. The leaf nodes aquire the
index values that point to the appropriate matrix-element entry.

We recursively scan the trie. Once we arrive at a leaf node, the index is
increased and associated to that node. Finally, we allocate the matrix-element
array with the appropriate size.

If matrix element values are temporarily stored within the leaf nodes, we scan the state again and transfer them to the matrix-element array.

⟨*State matrices: public*⟩+≡

```
    public :: state_matrix_freeze
```

⟨*State matrices: interfaces*⟩≡

```
    interface state_matrix_freeze
       module procedure state_matrix_freeze1
       module procedure state_matrix_freeze2
    end interface
```

⟨*State matrices: procedures*⟩+≡

```
    subroutine state_matrix_freeze1 (state)
      type(state_matrix_t), intent(inout), target :: state
      type(state_iterator_t) :: it
      if (associated (state%root)) then
         if (allocated (state%me))  deallocate (state%me)
         allocate (state%me (state%n_matrix_elements))
         state%me = 0
      end if
      if (state%leaf_nodes_store_values) then
         call state_iterator_init (it, state)
         do while (state_iterator_is_valid (it))
            state%me(state_iterator_get_me_index (it)) &
                 = state_iterator_get_matrix_element (it)
            call state_iterator_advance (it)
         end do
         state%leaf_nodes_store_values = .false.
      end if
    end subroutine state_matrix_freeze1

    subroutine state_matrix_freeze2 (state)
      type(state_matrix_t), dimension(:), intent(inout), target :: state
      integer :: i
      do i = 1, size (state)
         call state_matrix_freeze1 (state(i))
      end do
    end subroutine state_matrix_freeze2
```

Direct access to the value array

Several methods for setting a value directly are summarized in this generic:

⟨*State matrices: public*⟩+≡

```
    public :: state_matrix_set_matrix_element
```

⟨*State matrices: interfaces*⟩+≡

```
    interface state_matrix_set_matrix_element
       module procedure state_matrix_set_matrix_element_qn
       module procedure state_matrix_set_matrix_element_all
       module procedure state_matrix_set_matrix_element_array
       module procedure state_matrix_set_matrix_element_single
       module procedure state_matrix_set_matrix_element_clone
    end interface
```

Set a value that corresponds to a quantum number array:

⟨State matrices: procedures⟩+≡

```
subroutine state_matrix_set_matrix_element_qn (state, qn, value)
  type(state_matrix_t), intent(inout) :: state
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  complex(default), intent(in) :: value
  type(state_iterator_t) :: it
  if (.not. allocated (it%state%me)) then
     allocate (it%state%me (size(qn)))
  end if
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     if (all (qn == state_iterator_get_quantum_numbers (it))) then
        call state_iterator_set_matrix_element (it, value)
        return
     end if
     call state_iterator_advance (it)
  end do
end subroutine state_matrix_set_matrix_element_qn
```

Set all matrix elements to a single value

⟨State matrices: procedures⟩+≡

```
subroutine state_matrix_set_matrix_element_all (state, value)
  type(state_matrix_t), intent(inout) :: state
  complex(default), intent(in) :: value
  if (.not. allocated (state%me)) then
     allocate (state%me (state%n_matrix_elements))
  end if
  state%me = value
end subroutine state_matrix_set_matrix_element_all
```

Set the matrix-element array directly.

⟨State matrices: procedures⟩+≡

```
subroutine state_matrix_set_matrix_element_array (state, value)
  type(state_matrix_t), intent(inout) :: state
  complex(default), dimension(:), intent(in) :: value
  if (.not. allocated (state%me)) then
     allocate (state%me (size (value)))
  end if
  state%me = value
end subroutine state_matrix_set_matrix_element_array

pure subroutine state_matrix_set_matrix_element_single (state, i, value)
  type(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  complex(default), intent(in) :: value
  if (.not. allocated (state%me)) then
     allocate (state%me (state%n_matrix_elements))
  end if
  state%me(i) = value
end subroutine state_matrix_set_matrix_element_single
```

Clone the matrix elements from another (matching) state matrix.

⟨State matrices: procedures⟩+≡
```
    subroutine state_matrix_set_matrix_element_clone (state, state1)
      type(state_matrix_t), intent(inout) :: state
      type(state_matrix_t), intent(in) :: state1
      if (.not. allocated (state1%me)) return
      if (.not. allocated (state%me)) allocate (state%me (size (state1%me)))
      state%me = state1%me
    end subroutine state_matrix_set_matrix_element_clone
```

Add a value to a matrix element

⟨State matrices: public⟩+≡
```
    public :: state_matrix_add_to_matrix_element
```

⟨State matrices: procedures⟩+≡
```
    subroutine state_matrix_add_to_matrix_element (state, i, value)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      complex(default), intent(in) :: value
      state%me(i) = state%me(i) + value
    end subroutine state_matrix_add_to_matrix_element
```

7.5.3 State iterators

Accessing the quantum state from outside is best done using a specialized itera-
tor, i.e., a pointer to a particular branch of the quantum state trie. Technically,
the iterator contains a pointer to a leaf node, but via parent pointers it allows
to access the whole branch where the leaf is attached. For quick access, we also
keep the branch depth (which is assumed to be universal for a quantum state).

⟨State matrices: public⟩+≡
```
    public :: state_iterator_t
```

⟨State matrices: types⟩+≡
```
    type :: state_iterator_t
      private
      integer :: depth = 0
      type(state_matrix_t), pointer :: state => null ()
      type(node_t), pointer :: node => null ()
    end type state_iterator_t
```

The initializer: Point at the first branch. Note that this cannot be pure, thus
not be elemental, because the iterator can be used to manipulate data in the
state matrix.

⟨State matrices: public⟩+≡
```
    public :: state_iterator_init
```

⟨State matrices: procedures⟩+≡
```
    subroutine state_iterator_init (it, state)
      type(state_iterator_t), intent(out) :: it
      type(state_matrix_t), intent(in), target :: state
      it%state => state
      it%depth = state%depth
```

```
      if (state_matrix_is_defined (state)) then
         it%node => state%root
         do while (associated (it%node%child_first))
            it%node => it%node%child_first
         end do
      else
         it%node => null ()
      end if
   end subroutine state_iterator_init
```

Go forward. Recursively programmed: if the next node does not exist, go back to the parent node and look at its successor (if present), etc.

There is a possible pitfall in the implementation: If the dummy pointer argument to the `find_next` routine is used directly, we still get the correct result for the iterator, but calling the recursion on `node%parent` means that we manipulate a parent pointer in the original state in addition to the iterator. Making a local copy of the pointer avoids this. Using pointer intent would be helpful, but we do not yet rely on this F2003 feature.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_advance
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_iterator_advance (it)
      type(state_iterator_t), intent(inout) :: it
      call find_next (it%node)
    contains
      recursive subroutine find_next (node_in)
        type(node_t), intent(in), target :: node_in
        type(node_t), pointer :: node
        node => node_in
        if (associated (node%next)) then
           node => node%next
           do while (associated (node%child_first))
              node => node%child_first
           end do
           it%node => node
        else if (associated (node%parent)) then
           call find_next (node%parent)
        else
           it%node => null ()
        end if
      end subroutine find_next
    end subroutine state_iterator_advance
```

If all has been scanned, the iterator is at an undefined state. Check for this:

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_is_valid
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_is_valid (it) result (defined)
      logical :: defined
      type(state_iterator_t), intent(in) :: it
      defined = associated (it%node)
```

```
      end function state_iterator_is_valid
```

Return the matrix-element index that corresponds to the current node

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_me_index
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_get_me_index (it) result (n)
      integer :: n
      type(state_iterator_t), intent(in) :: it
      n = it%node%me_index
    end function state_iterator_get_me_index
```

Return the number of times this quantum-number state has been added (noting that it is physically inserted only the first time). Note that for each state, there is an array of counters.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_me_count
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_get_me_count (it) result (n)
      integer, dimension(:), allocatable :: n
      type(state_iterator_t), intent(in) :: it
      if (allocated (it%node%me_count)) then
         allocate (n (size (it%node%me_count)))
         n = it%node%me_count
      else
         allocate (n (0))
      end if
    end function state_iterator_get_me_count
```

Use the iterator to retrieve quantum-number information:

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_quantum_numbers
    public :: state_iterator_get_flavor
    public :: state_iterator_get_color
    public :: state_iterator_get_helicity
```

⟨*State matrices: interfaces*⟩+≡
```
    interface state_iterator_get_quantum_numbers
       module procedure state_iterator_get_qn_multi
       module procedure state_iterator_get_qn_slice
       module procedure state_iterator_get_qn_range
       module procedure state_iterator_get_qn_single
    end interface

    interface state_iterator_get_flavor
       module procedure state_iterator_get_flv_multi
       module procedure state_iterator_get_flv_slice
       module procedure state_iterator_get_flv_range
       module procedure state_iterator_get_flv_single
    end interface
```

```
interface state_iterator_get_color
   module procedure state_iterator_get_col_multi
   module procedure state_iterator_get_col_slice
   module procedure state_iterator_get_col_range
   module procedure state_iterator_get_col_single
end interface

interface state_iterator_get_helicity
   module procedure state_iterator_get_hel_multi
   module procedure state_iterator_get_hel_slice
   module procedure state_iterator_get_hel_range
   module procedure state_iterator_get_hel_single
end interface
```

These versions return the whole quantum number array

⟨*State matrices: procedures*⟩+≡

```
function state_iterator_get_qn_multi (it) result (qn)
  type(state_iterator_t), intent(in) :: it
  type(quantum_numbers_t), dimension(it%depth) :: qn
  type(node_t), pointer :: node
  integer :: i
  node => it%node
  do i = it%depth, 1, -1
     qn(i) = node%qn
     node => node%parent
  end do
end function state_iterator_get_qn_multi

function state_iterator_get_flv_multi (it) result (flv)
  type(state_iterator_t), intent(in) :: it
  type(flavor_t), dimension(it%depth) :: flv
  flv = quantum_numbers_get_flavor &
       (state_iterator_get_quantum_numbers (it))
end function state_iterator_get_flv_multi

function state_iterator_get_col_multi (it) result (col)
  type(state_iterator_t), intent(in) :: it
  type(color_t), dimension(it%depth) :: col
  col = quantum_numbers_get_color &
       (state_iterator_get_quantum_numbers (it))
end function state_iterator_get_col_multi

function state_iterator_get_hel_multi (it) result (hel)
  type(state_iterator_t), intent(in) :: it
  type(helicity_t), dimension(it%depth) :: hel
  hel = quantum_numbers_get_helicity &
       (state_iterator_get_quantum_numbers (it))
end function state_iterator_get_hel_multi
```

An array slice (derived from the above).

⟨*State matrices: procedures*⟩+≡

```
function state_iterator_get_qn_slice (it, index) result (qn)
  type(state_iterator_t), intent(in) :: it
```

769

```fortran
    integer, dimension(:), intent(in) :: index
    type(quantum_numbers_t), dimension(size(index)) :: qn
    type(quantum_numbers_t), dimension(it%depth) :: qn_tmp
    qn_tmp = state_iterator_get_qn_multi (it)
    qn = qn_tmp(index)
  end function state_iterator_get_qn_slice

  function state_iterator_get_flv_slice (it, index) result (flv)
    type(state_iterator_t), intent(in) :: it
    integer, dimension(:), intent(in) :: index
    type(flavor_t), dimension(size(index)) :: flv
    flv = quantum_numbers_get_flavor &
         (state_iterator_get_quantum_numbers (it, index))
  end function state_iterator_get_flv_slice

  function state_iterator_get_col_slice (it, index) result (col)
    type(state_iterator_t), intent(in) :: it
    integer, dimension(:), intent(in) :: index
    type(color_t), dimension(size(index)) :: col
    col = quantum_numbers_get_color &
         (state_iterator_get_quantum_numbers (it, index))
  end function state_iterator_get_col_slice

  function state_iterator_get_hel_slice (it, index) result (hel)
    type(state_iterator_t), intent(in) :: it
    integer, dimension(:), intent(in) :: index
    type(helicity_t), dimension(size(index)) :: hel
    hel = quantum_numbers_get_helicity &
         (state_iterator_get_quantum_numbers (it, index))
  end function state_iterator_get_hel_slice
```

An array range (implemented directly).

⟨*State matrices: procedures*⟩+≡

```fortran
  function state_iterator_get_qn_range (it, k1, k2) result (qn)
    type(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k1, k2
    type(quantum_numbers_t), dimension(k2-k1+1) :: qn
    type(node_t), pointer :: node
    integer :: i
    node => it%node
    SCAN: do i = it%depth, 1, -1
       if (k1 <= i .and. i <= k2) then
          qn(i-k1+1) = node%qn
       else
          node => node%parent
       end if
    end do SCAN
  end function state_iterator_get_qn_range

  function state_iterator_get_flv_range (it, k1, k2) result (flv)
    type(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k1, k2
    type(flavor_t), dimension(k2-k1+1) :: flv
    flv = quantum_numbers_get_flavor &
```

```
            (state_iterator_get_quantum_numbers (it, k1, k2))
    end function state_iterator_get_flv_range

    function state_iterator_get_col_range (it, k1, k2) result (col)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k1, k2
      type(color_t), dimension(k2-k1+1) :: col
      col = quantum_numbers_get_color &
            (state_iterator_get_quantum_numbers (it, k1, k2))
    end function state_iterator_get_col_range

    function state_iterator_get_hel_range (it, k1, k2) result (hel)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k1, k2
      type(helicity_t), dimension(k2-k1+1) :: hel
      hel = quantum_numbers_get_helicity &
            (state_iterator_get_quantum_numbers (it, k1, k2))
    end function state_iterator_get_hel_range
```

Just a specific single element

⟨*State matrices: procedures*⟩+≡

```
    function state_iterator_get_qn_single (it, k) result (qn)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k
      type(quantum_numbers_t) :: qn
      type(node_t), pointer :: node
      integer :: i
      node => it%node
      SCAN: do i = it%depth, 1, -1
         if (i == k) then
            qn = node%qn
            exit SCAN
         else
            node => node%parent
         end if
      end do SCAN
    end function state_iterator_get_qn_single

    function state_iterator_get_flv_single (it, k) result (flv)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k
      type(flavor_t) :: flv
      flv = quantum_numbers_get_flavor &
            (state_iterator_get_quantum_numbers (it, k))
    end function state_iterator_get_flv_single

    function state_iterator_get_col_single (it, k) result (col)
      type(state_iterator_t), intent(in) :: it
      integer, intent(in) :: k
      type(color_t) :: col
      col = quantum_numbers_get_color &
            (state_iterator_get_quantum_numbers (it, k))
    end function state_iterator_get_col_single
```

```
      function state_iterator_get_hel_single (it, k) result (hel)
        type(state_iterator_t), intent(in) :: it
        integer, intent(in) :: k
        type(helicity_t) :: hel
        hel = quantum_numbers_get_helicity &
             (state_iterator_get_quantum_numbers (it, k))
      end function state_iterator_get_hel_single
```

Retrieve the matrix element value associated with the current node.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_get_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    function state_iterator_get_matrix_element (it) result (me)
      complex(default) :: me
      type(state_iterator_t), intent(in) :: it
      if (it%state%leaf_nodes_store_values) then
         me = it%node%me
      else if (it%node%me_index /= 0) then
         me = it%state%me(it%node%me_index)
      else
         me = 0
      end if
    end function state_iterator_get_matrix_element
```

Set the matrix element value using the state iterator.

⟨*State matrices: public*⟩+≡
```
    public :: state_iterator_set_matrix_element
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_iterator_set_matrix_element (it, value)
      type(state_iterator_t), intent(inout) :: it
      complex(default), intent(in) :: value
      if (it%node%me_index /= 0) then
         it%state%me(it%node%me_index) = value
      end if
    end subroutine state_iterator_set_matrix_element
```

7.5.4   Operations on quantum states

Return a deep copy of a state matrix.

⟨*State matrices: public*⟩+≡
```
    public :: assignment(=)
```

⟨*State matrices: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure state_matrix_assign
    end interface
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_assign (state_out, state_in)
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), intent(in), target :: state_in
  type(state_iterator_t) :: it
  if (.not. state_matrix_is_defined (state_in))  return
  call state_matrix_init (state_out)
  call state_iterator_init (it, state_in)
  do while (state_iterator_is_valid (it))
     call state_matrix_add_state (state_out, &
           state_iterator_get_quantum_numbers (it), &
           state_iterator_get_me_index (it))
     call state_iterator_advance (it)
  end do
  if (allocated (state_in%me)) then
     allocate (state_out%me (size (state_in%me)))
     state_out%me = state_in%me
  end if
end subroutine state_matrix_assign
```

Determine the indices of all diagonal matrix elements.

⟨*State matrices: public*⟩+≡

```
public :: state_matrix_get_diagonal_entries
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_get_diagonal_entries (state, i)
  type(state_matrix_t), intent(in) :: state
  integer, dimension(:), allocatable, intent(out) :: i
  integer, dimension(state%n_matrix_elements) :: tmp
  integer :: n
  type(state_iterator_t) :: it
  n = 0
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     if (all (quantum_numbers_are_diagonal ( &
           state_iterator_get_quantum_numbers (it)))) then
        n = n + 1
        tmp(n) = state_iterator_get_me_index (it)
     end if
     call state_iterator_advance (it)
  end do
  allocate (i(n))
  if (n > 0) i = tmp(:n)
end subroutine state_matrix_get_diagonal_entries
```

Normalize all matrix elements, i.e., multiply by a common factor. Assuming that the factor is nonzero, of course.

⟨*State matrices: public*⟩+≡

```
public :: state_matrix_renormalize
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_renormalize (state, factor)
  type(state_matrix_t), intent(inout) :: state
  complex(default), intent(in) :: factor
```

```
      state%me = state%me * factor
    end subroutine state_matrix_renormalize
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is
reflected in the state-matrix norm.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_normalize_by_trace
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_normalize_by_trace (state)
      type(state_matrix_t), intent(inout) :: state
      real(default) :: trace
      trace = state_matrix_trace (state)
      if (trace /= 0) then
         state%me = state%me / trace
         state%norm = state%norm * trace
      end if
    end subroutine state_matrix_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_normalize_by_max
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_normalize_by_max (state)
      type(state_matrix_t), intent(inout) :: state
      real(default) :: m
      m = maxval (abs (state%me))
      if (m /= 0) then
         state%me = state%me / m
         state%norm = state%norm * m
      end if
    end subroutine state_matrix_normalize_by_max
```

Explicitly set the norm of a state matrix.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_set_norm
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_set_norm (state, norm)
      type(state_matrix_t), intent(inout) :: state
      real(default), intent(in) :: norm
      state%norm = norm
    end subroutine state_matrix_set_norm
```

Return the sum of all matrix element values.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_sum
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_sum (state) result (value)
      complex(default) :: value
      type(state_matrix_t), intent(in) :: state
```

```
      value = sum (state%me)
    end function state_matrix_sum
```

Return the trace of a state matrix, i.e., the sum over all diagonal values. If
`qn_in` is provided, only branches that match this quantum-numbers array are
considered.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_trace
```

⟨*State matrices: procedures*⟩+≡
```
    function state_matrix_trace (state, qn_in) result (trace)
      complex(default) :: trace
      type(state_matrix_t), intent(in), target :: state
      type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      type(state_iterator_t) :: it
      allocate (qn (state_matrix_get_depth (state)))
      trace = 0
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         qn = state_iterator_get_quantum_numbers (it)
         if (present (qn_in)) then
            if (.not. all (qn .match. qn_in)) then
               call state_iterator_advance (it);   cycle
            end if
         end if
         if (all (quantum_numbers_are_diagonal (qn))) then
            trace = trace + state_iterator_get_matrix_element (it)
         end if
         call state_iterator_advance (it)
      end do
    end function state_matrix_trace
```

Append new states which are color-contracted versions of the existing states.
The matrix element index of each color contraction coincides with the index of
its origin, so no new matrix elements are generated. After this operation, no
`freeze` must be performed anymore.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_add_color_contractions
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_add_color_contractions (state)
      type(state_matrix_t), intent(inout), target :: state
      type(state_iterator_t) :: it
      type(quantum_numbers_t), dimension(:,:), allocatable :: qn
      type(quantum_numbers_t), dimension(:,:), allocatable :: qn_con
      integer, dimension(:), allocatable :: me_index
      integer :: depth, n_me, i, j
      depth = state_matrix_get_depth (state)
      n_me = state_matrix_get_n_matrix_elements (state)
      allocate (qn (depth, n_me))
      allocate (me_index (n_me))
      i = 0
```

```
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         i = i + 1
         qn(:,i) = state_iterator_get_quantum_numbers (it)
         me_index(i) = state_iterator_get_me_index (it)
         call state_iterator_advance (it)
      end do
      do i = 1, n_me
         call quantum_number_array_make_color_contractions (qn(:,i), qn_con)
         do j = 1, size (qn_con, 2)
            call state_matrix_add_state (state, qn_con(:,j), index = me_index(i))
         end do
      end do
   end subroutine state_matrix_add_color_contractions
```

This procedure merges two state matrices of equal depth. For each quantum
number (flavor, color, helicity), we take the entry from the first argument where
defined, otherwise the second one. (If both are defined, we get an off-diagonal
matrix.) The resulting trie combines the information of the input tries in all
possible ways. Note that values are ignored, all values in the result are zero.

⟨*State matrices: public*⟩+≡
```
   public :: merge_state_matrices
```

⟨*State matrices: procedures*⟩+≡
```
   subroutine merge_state_matrices (state1, state2, state3)
     type(state_matrix_t), intent(in), target :: state1, state2
     type(state_matrix_t), intent(out) :: state3
     type(state_iterator_t) :: it1, it2
     type(quantum_numbers_t), dimension(state1%depth) :: qn1, qn2
     if (state1%depth /= state2%depth) then
        call state_matrix_write (state1)
        call state_matrix_write (state2)
        call msg_bug ("State matrices merge impossible: incompatible depths")
     end if
     call state_matrix_init (state3)
     call state_iterator_init (it1, state1)
     do while (state_iterator_is_valid (it1))
        qn1 = state_iterator_get_quantum_numbers (it1)
        call state_iterator_init (it2, state2)
        do while (state_iterator_is_valid (it2))
           qn2 = state_iterator_get_quantum_numbers (it2)
           call state_matrix_add_state &
                (state3, qn1 .merge. qn2)
           call state_iterator_advance (it2)
        end do
        call state_iterator_advance (it1)
     end do
     call state_matrix_freeze (state3)
   end subroutine merge_state_matrices
```

Multiply matrix elements from two state matrices. Choose the elements as
given by the integer index arrays, multiply them and store the sum of products

776

in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

Note that the `dot_product` intrinsic function conjugates its first complex argument. This is intended for the `c` suffix case, but must be reverted for the plain-product case.

We provide analogous subroutines for just summing over state matrix entries. The `evaluate_sum` variant includes the state-matrix norm in the evaluation, the `evaluate_me_sum` takes into account just the matrix elements proper.

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_evaluate_product
    public :: state_matrix_evaluate_product_cf
    public :: state_matrix_evaluate_square_c
    public :: state_matrix_evaluate_sum
    public :: state_matrix_evaluate_me_sum
```

⟨*State matrices: procedures*⟩+≡
```
    pure subroutine state_matrix_evaluate_product &
        (state, i, state1, state2, index1, index2)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1, state2
      integer, dimension(:), intent(in) :: index1, index2
      state%me(i) = &
          dot_product (conjg (state1%me(index1)), state2%me(index2))
      state%norm = state1%norm * state2%norm
    end subroutine state_matrix_evaluate_product

    pure subroutine state_matrix_evaluate_product_cf &
        (state, i, state1, state2, index1, index2, factor)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1, state2
      integer, dimension(:), intent(in) :: index1, index2
      complex(default), dimension(:), intent(in) :: factor
      state%me(i) = &
          dot_product (state1%me(index1), factor * state2%me(index2))
      state%norm = state1%norm * state2%norm
    end subroutine state_matrix_evaluate_product_cf

    pure subroutine state_matrix_evaluate_square_c (state, i, state1, index1)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1
      integer, dimension(:), intent(in) :: index1
      state%me(i) = &
          dot_product (state1%me(index1), state1%me(index1))
      state%norm = abs (state1%norm) ** 2
    end subroutine state_matrix_evaluate_square_c

    pure subroutine state_matrix_evaluate_sum (state, i, state1, index1)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1
      integer, dimension(:), intent(in) :: index1
```

```
      state%me(i) = &
           sum (state1%me(index1)) * state1%norm
    end subroutine state_matrix_evaluate_sum

    pure subroutine state_matrix_evaluate_me_sum (state, i, state1, index1)
      type(state_matrix_t), intent(inout) :: state
      integer, intent(in) :: i
      type(state_matrix_t), intent(in) :: state1
      integer, dimension(:), intent(in) :: index1
      state%me(i) = sum (state1%me(index1))
    end subroutine state_matrix_evaluate_me_sum
```

Outer product (of states and matrix elements):

⟨*State matrices: public*⟩+≡
```
    public :: outer_multiply
```

⟨*State matrices: interfaces*⟩+≡
```
    interface outer_multiply
       module procedure outer_multiply_pair
       module procedure outer_multiply_array
    end interface
```

This procedure constructs the outer product of two state matrices.

⟨*State matrices: procedures*⟩+≡
```
    subroutine outer_multiply_pair (state1, state2, state3)
      type(state_matrix_t), intent(in), target :: state1, state2
      type(state_matrix_t), intent(out) :: state3
      type(state_iterator_t) :: it1, it2
      type(quantum_numbers_t), dimension(state1%depth) :: qn1
      type(quantum_numbers_t), dimension(state2%depth) :: qn2
      type(quantum_numbers_t), dimension(state1%depth+state2%depth) :: qn3
      complex(default) :: val1, val2
      call state_matrix_init (state3, store_values=.true.)
      call state_iterator_init (it1, state1)
      do while (state_iterator_is_valid (it1))
         qn1 = state_iterator_get_quantum_numbers (it1)
         val1 = state_iterator_get_matrix_element (it1)
         call state_iterator_init (it2, state2)
         do while (state_iterator_is_valid (it2))
            qn2 = state_iterator_get_quantum_numbers (it2)
            val2 = state_iterator_get_matrix_element (it2)
            qn3(:state1%depth) = qn1
            qn3(state1%depth+1:) = qn2
            call state_matrix_add_state (state3, qn3, value=val1 * val2)
            call state_iterator_advance (it2)
         end do
         call state_iterator_advance (it1)
      end do
      call state_matrix_freeze (state3)
    end subroutine outer_multiply_pair
```

This executes the above routine iteratively for an arbitrary number of state
matrices.

778

⟨*State matrices: procedures*⟩+≡

```
subroutine outer_multiply_array (state_in, state_out)
  type(state_matrix_t), dimension(:), intent(in), target :: state_in
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), dimension(:), allocatable, target :: state_tmp
  integer :: i, n
  n = size (state_in)
  select case (n)
  case (0)
     call state_matrix_init (state_out)
  case (1)
     state_out = state_in(1)
  case (2)
     call outer_multiply_pair (state_in(1), state_in(2), state_out)
  case default
     allocate (state_tmp (n-2))
     call outer_multiply_pair (state_in(1), state_in(2), state_tmp(1))
     do i = 2, n - 2
        call outer_multiply_pair (state_tmp(i-1), state_in(i+1), state_tmp(i))
     end do
     call outer_multiply_pair (state_tmp(n-2), state_in(n), state_out)
     call state_matrix_final (state_tmp)
  end select
end subroutine outer_multiply_array
```

7.5.5   Factorization

In physical events, the state matrix is factorized into single-particle state matrices. This is essentially a measurement.

In a simulation, we select one particular branch of the state matrix with a probability that is determined by the matrix elements at the leaves. (This makes sense only if the state matrix represents a squared amplitude.) The selection is based on a (random) value x between 0 and one that is provided as the third argument.

For flavor and color, we select a unique value for each particle. For polarization, we have three options (modes). Option 1 is to drop helicity information altogether and sum over all diagonal helicities. Option 2 is to select a unique diagonal helicity in the same way as flavor and color. Option 3 is, for each particle, to trace over all remaining helicities in order to obtain an array of independent single-particle helicity matrices.

Only branches that match the given quantum-number array qn_in, if present, are considered. For this array, color is ignored.

If the optional correlated_state is provided, it is assigned the correlated density matrix for the selected flavor-color branch, so multi-particle spin correlations remain available even if they are dropped in the single-particle density matrices.

The algorithm is as follows: First, we determine the normalization by summing over all diagonal matrix elements. In a second scan, we select one of the diagonal matrix elements by a cumulative comparison with the normalized random number. In the corresponding quantum number array, we undefine the

helicity entries. Then, we scan the third time. For each branch that matches the selected quantum number array (i.e., definite flavor and color, arbitrary helicity), we determine its contribution to any of the single-particle state matrices. The matrix-element value is added if all other quantum numbers are diagonal, while the helicity of the chosen particle may be arbitrary; this helicity determines the branch in the single-particle state.

As a result, flavor and color quantum numbers are selected with the correct probability. Within this subset of states, each single-particle state matrix results from tracing over all other particles. Note that the single-particle state matrices are not normalized.

The flag ok is set to false if the matrix element sum is zero, so factorization is not possible. This can happen if an event did not pass cuts.

⟨*State matrices: parameters*⟩≡
```
integer, parameter, public :: FM_IGNORE_HELICITY = 1
integer, parameter, public :: FM_SELECT_HELICITY = 2
integer, parameter, public :: FM_FACTOR_HELICITY = 3
```

⟨*State matrices: public*⟩+≡
```
public :: state_matrix_factorize
```

⟨*State matrices: procedures*⟩+≡
```
subroutine state_matrix_factorize &
     (state, mode, x, ok, single_state, correlated_state, qn_in)
  type(state_matrix_t), intent(in), target :: state
  integer, intent(in) :: mode
  real(default), intent(in) :: x
  logical, intent(out) :: ok
  type(state_matrix_t), &
       dimension(:), allocatable, intent(out) :: single_state
  type(state_matrix_t), intent(out), optional :: correlated_state
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
  type(state_iterator_t) :: it
  real(default) :: s, xt
  complex(default) :: value
  integer :: i, depth
  type(quantum_numbers_t), dimension(:), allocatable :: qn, qn1
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  logical, dimension(:), allocatable :: diagonal
  logical, dimension(:,:), allocatable :: mask
  ok = .true.
  if (x /= 0) then
     xt = x * state_matrix_trace (state, qn_in)
  else
     xt = 0
  end if
  s = 0
  depth = state_matrix_get_depth (state)
  allocate (qn (depth), qn1 (depth), diagonal (depth))
  call state_iterator_init (it, state)
  do while (state_iterator_is_valid (it))
     qn = state_iterator_get_quantum_numbers (it)
     if (present (qn_in)) then
        if (.not. all (qn .fhmatch. qn_in)) then
```

780

```
            call state_iterator_advance (it); cycle
         end if
      end if
      if (all (quantum_numbers_are_diagonal (qn))) then
         value = state_iterator_get_matrix_element (it)
         if (real (value, default) < 0) then
            call state_matrix_write (state)
            print *, value
            call msg_bug ("Event generation: " &
                 // "Negative real part of squared matrix element value")
            value = 0
         end if
         s = s + value
         if (s > xt)  exit
      end if
      call state_iterator_advance (it)
   end do
   if (.not. state_iterator_is_valid (it)) then
      if (s == 0)  ok = .false.
      call state_iterator_init (it, state)
   end if
   allocate (single_state (depth))
   call state_matrix_init (single_state, store_values=.true.)
   if (present (correlated_state)) &
        call state_matrix_init (correlated_state, store_values=.true.)
   qn = state_iterator_get_quantum_numbers (it)
   select case (mode)
   case (FM_SELECT_HELICITY)  ! single branch selected; shortcut
      do i = 1, depth
         call state_matrix_add_state (single_state(i), &
              (/qn(i)/), value=value)
      end do
      if (.not. present (correlated_state)) then
         call state_matrix_freeze (single_state)
         return
      end if
   end select
   allocate (qn_mask (depth))
   call quantum_numbers_mask_init (qn_mask, .false., .false., .false., .true.)
   call quantum_numbers_undefine (qn, qn_mask)
   select case (mode)
   case (FM_FACTOR_HELICITY)
      allocate (mask (depth, depth))
      mask = .false.
      forall (i = 1:depth)  mask(i,i) = .true.
   end select
   call state_iterator_init (it, state)
   do while (state_iterator_is_valid (it))
      qn1 = state_iterator_get_quantum_numbers (it)
      if (all (qn .match. qn1)) then
         diagonal = quantum_numbers_are_diagonal (qn1)
         value = state_iterator_get_matrix_element (it)
         select case (mode)
         case (FM_IGNORE_HELICITY)  ! trace over diagonal states that match qn
```

```
                  if (all (diagonal)) then
                     do i = 1, depth
                        call state_matrix_add_state (single_state(i), &
                             (/qn(i)/), value=value, sum_values=.true.)
                     end do
                  end if
               case (FM_FACTOR_HELICITY)  ! trace over all other particles
                  do i = 1, depth
                     if (all (diagonal .or. mask(:,i))) then
                        call state_matrix_add_state (single_state(i), &
                             (/qn1(i)/), value=value, sum_values=.true.)
                     end if
                  end do
               end select
               if (present (correlated_state)) &
                    call state_matrix_add_state (correlated_state, qn1, value=value)
            end if
            call state_iterator_advance (it)
         end do
         call state_matrix_freeze (single_state)
         if (present (correlated_state)) &
              call state_matrix_freeze (correlated_state)
    end subroutine state_matrix_factorize
```

⟨*State matrices: procedures*⟩+≡
```
    elemental function pacify_complex (c_in) result (c_pac)
      complex(default), intent(in) :: c_in
      complex(default) :: c_pac
      c_pac = c_in
      if (real(c_pac) == -real(c_pac)) then
         c_pac = &
              cmplx (0._default, aimag(c_pac), kind=default)
      end if
      if (aimag(c_pac) == -aimag(c_pac)) then
         c_pac = &
              cmplx (real(c_pac), 0._default, kind=default)
      end if
    end function pacify_complex
```

7.5.6 Unit tests

⟨*State matrices: public*⟩+≡
```
    public :: state_matrix_test
```

⟨*State matrices: procedures*⟩+≡
```
    subroutine state_matrix_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
      ⟨State matrices: execute tests⟩
    end subroutine state_matrix_test
```

Create two quantum states of equal depth and merge them.

⟨*State matrices: execute tests*⟩≡
```
call test (state_matrix_1, "state_matrix_1", &
     "check merge of quantum states of equal depth", &
     u, results)
```

⟨*State matrices: tests*⟩≡
```
subroutine state_matrix_1 (u)
  integer, intent(in) :: u
  type(state_matrix_t) :: state1, state2, state3
  type(flavor_t), dimension(3) :: flv
  type(color_t), dimension(3) :: col
  type(helicity_t), dimension(3) :: hel
  type(quantum_numbers_t), dimension(3) :: qn

  write (u, "(A)")  "* Test output: state_matrix_1"
  write (u, "(A)")  "*   Purpose: create and merge two quantum states"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  write (u, "(A)")  "*  State matrix 1"
  write (u, "(A)")

  call state_matrix_init (state1)
  call flavor_init (flv, [1, 2, 11])
  call helicity_init (hel, [1, 1, 1])
  call quantum_numbers_init (qn, flv, hel)
  call state_matrix_add_state (state1, qn)
  call helicity_init (hel, [1, 1, 1], [-1, 1, -1])
  call quantum_numbers_init (qn, flv, hel)
  call state_matrix_add_state (state1, qn)
  call state_matrix_freeze (state1)
  call state_matrix_write (state1, u)

  write (u, "(A)")
  write (u, "(A)")  "*  State matrix 2"
  write (u, "(A)")

  call state_matrix_init (state2)
  call color_init (col(1), [501])
  call color_init (col(2), [-501])
  call color_init (col(3), [0])
  call helicity_init (hel, [-1, -1, 0])
  call quantum_numbers_init (qn, col, hel)
  call state_matrix_add_state (state2, qn)
  call color_init (col(3), [99])
  call helicity_init (hel, [-1, -1, 0])
  call quantum_numbers_init (qn, col, hel)
  call state_matrix_add_state (state2, qn)
  call state_matrix_freeze (state2)
  call state_matrix_write (state2, u)

  write (u, "(A)")
```

783

```
      write (u, "(A)")  "* Merge the state matrices"
      write (u, "(A)")

      call merge_state_matrices (state1, state2, state3)
      call state_matrix_write (state3, u)

      write (u, "(A)")
      write (u, "(A)")  "* Collapse the state matrix"
      write (u, "(A)")

      call state_matrix_collapse (state3, &
           new_quantum_numbers_mask (.false., .false., &
                                     (/.true.,.false.,.false./)))
      call state_matrix_write (state3, u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"
      write (u, "(A)")

      call state_matrix_final (state1)
      call state_matrix_final (state2)
      call state_matrix_final (state3)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: state_matrix_1"
      write (u, "(A)")

    end subroutine state_matrix_1
```

Create a correlated three-particle state matrix and factorize it.

⟨*State matrices: execute tests*⟩+≡
```
    call test (state_matrix_2, "state_matrix_2", &
         "check factorizing 3-particle state matrix", &
         u, results)
```

⟨*State matrices: tests*⟩+≡
```
    subroutine state_matrix_2 (u)
      integer, intent(in) :: u
      type(state_matrix_t) :: state
      type(state_matrix_t), dimension(:), allocatable :: single_state
      type(state_matrix_t) :: correlated_state
      complex(default) :: z, val
      complex(default), dimension(-1:1) :: v
      integer :: f, h11, h12, h21, h22, i, mode
      type(flavor_t), dimension(2) :: flv
      type(color_t), dimension(2) :: col
      type(helicity_t), dimension(2) :: hel
      type(quantum_numbers_t), dimension(2) :: qn
      logical :: ok

      write (u, "(A)")
      write (u, "(A)")  "* Test output: state_matrix_2"
      write (u, "(A)")  "*   Purpose: factorize correlated 3-particle state"
      write (u, "(A)")
```

```
write (u, "(A)")  "*  Initialization"
write (u, "(A)")

z = 1 / 2._default
v(-1) = (0.6_default, 0._default)
v( 1) = (0._default, 0.8_default)
call state_matrix_init (state)
do f = 1, 2
   do h11 = -1, 1, 2
      do h12 = -1, 1, 2
         do h21 = -1, 1, 2
            do h22 = -1, 1, 2
               call flavor_init (flv, [f, -f])
               call color_init (col(1), [1])
               call color_init (col(2), [-1])
               call helicity_init (hel, [h11,h12], [h21, h22])
               call quantum_numbers_init (qn, flv, col, hel)
               val = z * v(h11) * v(h12) * conjg (v(h21) * v(h22))
               call state_matrix_add_state (state, qn)
            end do
         end do
      end do
   end do
end do
call state_matrix_freeze (state)
call state_matrix_write (state, u)

write (u, "(A)")
write (u, "(A,'(',"  // FMT_19 // ",',','"  // FMT_19 // ",')')")") &
     "* Trace = ", state_matrix_trace (state)
write (u, "(A)")

do mode = 1, 3
   write (u, "(A)")
   write (u, "(A,I1)")  "* Mode = ", mode
   call state_matrix_factorize &
        (state, mode, 0.15_default, ok, single_state, correlated_state)
   do i = 1, size (single_state)
      write (u, "(A)")
      call state_matrix_write (single_state(i), u)
      write (u, "(A,'(',"  // FMT_19 // ",',','"  // FMT_19 // ",')')")") &
           "Trace = ", state_matrix_trace (single_state(i))
   end do
   write (u, "(A)")
   call state_matrix_write (correlated_state, u)
   write (u, "(A,'(',"  // FMT_19 // ",',','"  // FMT_19 // ",')')")")  &
        "Trace = ", state_matrix_trace (correlated_state)
   call state_matrix_final (single_state)
   call state_matrix_final (correlated_state)
end do

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
      call state_matrix_final (state)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: state_matrix_2"

   end subroutine state_matrix_2
```

Create a colored state matrix and add color contractions.

⟨*State matrices: execute tests*⟩+≡
```
      call test (state_matrix_3, "state_matrix_3", &
           "check factorizing 3-particle state matrix", &
           u, results)
```

⟨*State matrices: tests*⟩+≡
```
      subroutine state_matrix_3 (u)
        integer, intent(in) :: u
        type(state_matrix_t) :: state
        type(flavor_t), dimension(4) :: flv
        type(color_t), dimension(4) :: col
        type(quantum_numbers_t), dimension(4) :: qn

        write (u, "(A)")  "* Test output: state_matrix_3"
        write (u, "(A)")  "*   Purpose: add color connections to colored state"
        write (u, "(A)")

        write (u, "(A)")  "*  Initialization"
        write (u, "(A)")

        call state_matrix_init (state)
        call flavor_init (flv, &
             (/ 1, -HADRON_REMNANT_TRIPLET, -1, HADRON_REMNANT_TRIPLET /))
        call color_init (col(1), [17])
        call color_init (col(2), [-17])
        call color_init (col(3), [-19])
        call color_init (col(4), [19])
        call quantum_numbers_init (qn, flv=flv, col=col)
        call state_matrix_add_state (state, qn)
        call flavor_init (flv, &
             (/ 1, -HADRON_REMNANT_TRIPLET, 21, HADRON_REMNANT_OCTET /))
        call color_init (col(1), [17])
        call color_init (col(2), [-17])
        call color_init (col(3), [3, -5])
        call color_init (col(4), [5, -3])
        call quantum_numbers_init (qn, flv=flv, col=col)
        call state_matrix_add_state (state, qn)
        call state_matrix_freeze (state)

        write (u, "(A)") "* State:"
        write (u, "(A)")

        call state_matrix_write (state, u)
        call state_matrix_add_color_contractions (state)
```

786

```
      write (u, "(A)") "* State with contractions:"
      write (u, "(A)")

      call state_matrix_write (state, u)

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call state_matrix_final (state)

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: state_matrx_3"

    end subroutine state_matrix_3
```

7.6   Interactions

This module defines the `interaction_t` type. It is an extension of the `state_matrix_t`
type.

The state matrix is a representation of a multi-particle density matrix. It
implements all possible flavor, color, and quantum-number assignments of the
entries in a generic density matrix, and it can hold a complex matrix element
for each entry. (Note that this matrix can hold non-diagonal entries in color
and helicity space.) The `interaction_t` object associates this with a list of
momenta, such that the whole object represents a multi-particle state.

The `interaction_t` holds information about which particles are incoming,
virtual (i.e., kept for the records), or outgoing. Each particle can be associated
to a source within another interaction. This allows us to automatically fill those
interaction momenta which have been computed or defined elsewhere. It also
contains internal parent-child relations and flags for (virtual) particles which
are to be treated as resonances.

A quantum-number mask array summarizes, for each particle within the
interaction, the treatment of flavor, color, or helicity (expose or ignore). A list
of locks states which particles are bound to have an identical quantum-number
mask. This is useful when the mask is changed at one place.

⟨`interactions.f90`⟩≡
  ⟨*File header*⟩

```
  module interactions
```

  ⟨*Use kinds*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use sorting
    use subevents
    use expressions
    use flavors
    use colors
```

```
    use helicities
    use quantum_numbers
    use state_matrices
```

⟨*Standard module head*⟩

⟨*Interactions: public*⟩

⟨*Interactions: types*⟩

⟨*Interactions: interfaces*⟩

```
contains
```

⟨*Interactions: procedures*⟩

⟨*Interactions: tests*⟩

```
end module interactions
```

7.6.1 External interaction links

Each particle in an interaction can have a link to a corresponding particle in another interaction. This allows to fetch the momenta of incoming or virtual particles from the interaction where they are defined. The link object consists of a pointer to the interaction and an index.

⟨*Interactions: types*⟩≡
```
    type :: external_link_t
      private
      type(interaction_t), pointer :: int => null ()
      integer :: i
    end type external_link_t
```

Set an external link.

⟨*Interactions: procedures*⟩≡
```
    subroutine external_link_set (link, int, i)
      type(external_link_t), intent(out) :: link
      type(interaction_t), target, intent(in) :: int
      integer, intent(in) :: i
      if (i /= 0) then
         link%int => int
         link%i = i
      end if
    end subroutine external_link_set
```

Reassign an external link to a new interaction (which should be an image of the original target).

⟨*Interactions: procedures*⟩+≡
```
    subroutine external_link_reassign (link, int_src, int_target)
      type(external_link_t), intent(inout) :: link
      type(interaction_t), intent(in) :: int_src
      type(interaction_t), intent(in), target :: int_target
```

```
    if (associated (link%int)) then
       if (link%int%tag == int_src%tag)  link%int => int_target
    end if
end subroutine external_link_reassign
```

Return true if the link is set

⟨*Interactions: procedures*⟩+≡
```
function external_link_is_set (link) result (flag)
  logical :: flag
  type(external_link_t), intent(in) :: link
  flag = associated (link%int)
end function external_link_is_set
```

Return the interaction pointer.

⟨*Interactions: public*⟩≡
```
public :: external_link_get_ptr
```

⟨*Interactions: procedures*⟩+≡
```
function external_link_get_ptr (link) result (int)
  type(interaction_t), pointer :: int
  type(external_link_t), intent(in) :: link
  int => link%int
end function external_link_get_ptr
```

Return the index within that interaction

⟨*Interactions: public*⟩+≡
```
public :: external_link_get_index
```

⟨*Interactions: procedures*⟩+≡
```
function external_link_get_index (link) result (i)
  integer :: i
  type(external_link_t), intent(in) :: link
  i = link%i
end function external_link_get_index
```

Return a pointer to the momentum of the corresponding particle. If there is no
association, return a null pointer.

⟨*Interactions: procedures*⟩+≡
```
function external_link_get_momentum_ptr (link) result (p)
  type(vector4_t), pointer :: p
  type(external_link_t), intent(in) :: link
  if (associated (link%int)) then
     p => link%int%p(link%i)
  else
     p => null ()
  end if
end function external_link_get_momentum_ptr
```

7.6.2 Internal relations

In addition to the external links, particles within the interaction have parent-child relations. Here, more than one link is possible, and we set up an array.

⟨*Interactions: types*⟩+≡
```
type :: internal_link_list_t
  private
  integer :: length = 0
  integer, dimension(:), allocatable :: link
 contains
 ⟨Interactions: internal link list: TBP⟩
end type internal_link_list_t
```

Output, non-advancing.

⟨*Interactions: internal link list: TBP*⟩≡
```
procedure :: write => internal_link_list_write
```

⟨*Interactions: procedures*⟩+≡
```
subroutine internal_link_list_write (object, unit)
  class(internal_link_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit)
  do i = 1, object%length
     write (u, "(1x,I0)", advance="no")  object%link(i)
  end do
end subroutine internal_link_list_write
```

Append an item. Start with an array size of 2 and double the size if necessary.

⟨*Interactions: internal link list: TBP*⟩+≡
```
procedure :: append => internal_link_list_append
```

⟨*Interactions: procedures*⟩+≡
```
subroutine internal_link_list_append (link_list, link)
  class(internal_link_list_t), intent(inout) :: link_list
  integer, intent(in) :: link
  integer :: l
  integer, dimension(:), allocatable :: tmp
  l = link_list%length
  if (allocated (link_list%link)) then
     if (l == size (link_list%link)) then
        allocate (tmp (2 * l))
        tmp(:l) = link_list%link
        call move_alloc (from = tmp, to = link_list%link)
     end if
  else
     allocate (link_list%link (2))
  end if
  l = l + 1
  link_list%link(l) = link
  link_list%length = l
end subroutine internal_link_list_append
```

Return true if the link list is nonempty:

⟨Interactions: internal link list: TBP⟩+≡
```
    procedure :: has_entries => internal_link_list_has_entries
```

⟨Interactions: procedures⟩+≡
```
    function internal_link_list_has_entries (link_list) result (flag)
      class(internal_link_list_t), intent(in) :: link_list
      logical :: flag
      flag = link_list%length > 0
    end function internal_link_list_has_entries
```

Return the list length

⟨Interactions: internal link list: TBP⟩+≡
```
    procedure :: get_length => internal_link_list_get_length
```

⟨Interactions: procedures⟩+≡
```
    function internal_link_list_get_length (link_list) result (length)
      class(internal_link_list_t), intent(in) :: link_list
      integer :: length
      length = link_list%length
    end function internal_link_list_get_length
```

Return an entry.

⟨Interactions: internal link list: TBP⟩+≡
```
    procedure :: get_link => internal_link_list_get_link
```

⟨Interactions: procedures⟩+≡
```
    function internal_link_list_get_link (link_list, i) result (link)
      class(internal_link_list_t), intent(in) :: link_list
      integer, intent(in) :: i
      integer :: link
      if (i <= link_list%length) then
         link = link_list%link(i)
      else
         call msg_bug ("Internal link list: out of bounds")
      end if
    end function internal_link_list_get_link
```

7.6.3 The interaction type

An interaction is an entangled system of particles. Thus, the interaction object consists of two parts: the subevent, and the quantum state which technically is a trie. The subnode levels beyond the trie root node are in correspondence to the subevent, so both should be traversed in parallel.

The subevent is implemented as an allocatable array of four-momenta. The first `n_in` particles are incoming, `n_vir` particles in-between can be kept for bookkeeping, and the last `n_out` particles are outgoing.

Distinct interactions are linked by their particles: for each particle, we have the possibility of links to corresponding particles in other interactions. Furthermore, for bookkeeping purposes we have a self-link array `relations` where the parent-child relations are kept, and a flag array `resonant` which is set for an intermediate resonance.

Each momentum is associated with masks for flavor, color, and helicity. If a mask entry is set, the associated quantum number is to be ignored for that particle. If any mask has changed, the flag `update` is set.

We can have particle pairs locked together. If this is the case, the corresponding mask entries are bound to be equal. This is useful for particles that go through the interaction.

The interaction tag serves bookkeeping purposes. In particular, it identifies links in printout.

⟨*Interactions: public*⟩+≡
```
public :: interaction_t
```

⟨*Interactions: types*⟩+≡
```
type :: interaction_t
  private
  integer :: tag = 0
  type(state_matrix_t) :: state_matrix
  integer :: n_in = 0
  integer :: n_vir = 0
  integer :: n_out = 0
  integer :: n_tot = 0
  logical, dimension(:), allocatable :: p_is_known
  type(vector4_t), dimension(:), allocatable :: p
  type(external_link_t), dimension(:), allocatable :: source
  type(internal_link_list_t), dimension(:), allocatable :: parents
  type(internal_link_list_t), dimension(:), allocatable :: children
  logical, dimension(:), allocatable :: resonant
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
  integer, dimension(:), allocatable :: hel_lock
  logical :: update_state_matrix = .false.
  logical :: update_values = .false.
end type interaction_t
```

Initialize the particle array with a fixed size. The first `n_in` particles are incoming, the rest outgoing. Masks are optional. There is also an optional tag. The interaction still needs fixing the values, but that is to be done after all branches have been added.

Interaction tags are assigned consecutively, using a `save`d variable local to this procedure. If desired, we can provide a seed for the interaction tags. Such a seed should be positive. The default seed is one. `tag=0` indicates an empty interaction.

If `set_relations` is set and true, we establish parent-child relations for all incoming and outgoing particles. Virtual particles are skipped; this option is normally used only for interations without virtual particles.

⟨*Interactions: public*⟩+≡
```
public :: interaction_init
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_init &
    (int, n_in, n_vir, n_out, &
     tag, resonant, mask, hel_lock, set_relations, store_values)
  type(interaction_t), intent(out) :: int
  integer, intent(in) :: n_in, n_vir, n_out
  integer, intent(in), optional :: tag
```

```
      logical, dimension(:), intent(in), optional :: resonant
      type(quantum_numbers_mask_t), dimension(:), intent(in), optional :: mask
      integer, dimension(:), intent(in), optional :: hel_lock
      logical, intent(in), optional :: set_relations, store_values
      logical :: set_rel
      integer :: i, j
      set_rel = .false.;  if (present (set_relations))  set_rel = set_relations
      call interaction_set_tag (int, tag)
      call state_matrix_init (int%state_matrix, store_values)
      int%n_in = n_in
      int%n_vir = n_vir
      int%n_out = n_out
      int%n_tot = n_in + n_vir + n_out
      allocate (int%p_is_known (int%n_tot))
      int%p_is_known = .false.
      allocate (int%p (int%n_tot))
      allocate (int%source (int%n_tot))
      allocate (int%parents (int%n_tot))
      allocate (int%children (int%n_tot))
      allocate (int%resonant (int%n_tot))
      if (present (resonant)) then
         int%resonant = resonant
      else
         int%resonant = .false.
      end if
      allocate (int%mask (int%n_tot))
      allocate (int%hel_lock (int%n_tot))
      if (present (mask)) then
         int%mask = mask
      end if
      if (present (hel_lock)) then
         int%hel_lock = hel_lock
      else
         int%hel_lock = 0
      end if
      int%update_state_matrix = .false.
      int%update_values = .true.
      if (set_rel) then
         do i = 1, n_in
            do j = 1, n_out
               call interaction_relate (int, i, n_in + j)
            end do
         end do
      end if
    end subroutine interaction_init
```

Set or create a unique tag for the interaction. Without interaction, reset the tag counter.

⟨*Interactions: procedures*⟩+≡

```
    subroutine interaction_set_tag (int, tag)
      type(interaction_t), intent(inout), optional :: int
      integer, intent(in), optional :: tag
      integer, save :: stored_tag = 1
```

```
   if (present (int)) then
      if (present (tag)) then
         int%tag = tag
      else
         int%tag = stored_tag
         stored_tag = stored_tag + 1
      end if
   else if (present (tag)) then
      stored_tag = tag
   else
      stored_tag = 1
   end if
end subroutine interaction_set_tag
```

The public interface for the previous procedure only covers the reset functionality.

⟨*Interactions: public*⟩+≡
```
   public :: reset_interaction_counter
```

⟨*Interactions: procedures*⟩+≡
```
   subroutine reset_interaction_counter (tag)
     integer, intent(in), optional :: tag
     call interaction_set_tag (tag=tag)
   end subroutine reset_interaction_counter
```

Finalizer: The state-matrix object contains pointers.

⟨*Interactions: public*⟩+≡
```
   public :: interaction_final
```

⟨*Interactions: procedures*⟩+≡
```
   elemental subroutine interaction_final (int)
     type(interaction_t), intent(inout) :: int
     call state_matrix_final (int%state_matrix)
   end subroutine interaction_final
```

Output. The verbose option refers to the state matrix output.

⟨*Interactions: public*⟩+≡
```
   public :: interaction_write
```

⟨*Interactions: procedures*⟩+≡
```
   subroutine interaction_write &
        (int, unit, verbose, show_momentum_sum, show_mass, show_state, testflag)
     type(interaction_t), intent(in) :: int
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
     logical, intent(in), optional :: show_state, testflag
     integer :: u
     integer :: i, index_link
     type(interaction_t), pointer :: int_link
     logical :: show_st
     u = output_unit (unit);  if (u < 0)  return
     show_st = .true.;  if (present (show_state))  show_st = show_state
     if (int%tag /= 0) then
        write (u, "(1x,A,I0)")  "Interaction: ", int%tag
```

```
do i = 1, int%n_tot
   if (i == 1 .and. int%n_in > 0) then
      write (u, "(1x,A)") "Incoming:"
   else if (i == int%n_in + 1 .and. int%n_vir > 0) then
      write (u, "(1x,A)") "Virtual:"
   else if (i == int%n_in + int%n_vir + 1 .and. int%n_out > 0) then
      write (u, "(1x,A)") "Outgoing:"
   end if
   write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
   if (allocated (int%resonant)) then
      if (int%resonant(i)) then
         write (u, "(A)") "[r]"
      else
         write (u, *)
      end if
   else
      write (u, *)
   end if
   if (allocated (int%p)) then
      if (int%p_is_known(i)) then
         call vector4_write (int%p(i), u, show_mass, testflag)
      else
         write (u, "(A)")  "  [momentum undefined]"
      end if
   else
      write (u, "(A)") "  [momentum not allocated]"
   end if
   if (allocated (int%mask)) then
      write (u, "(1x,A)", advance="no")  "mask [fch] = "
      call quantum_numbers_mask_write (int%mask(i), u)
      write (u, *)
   end if
   if (int%parents(i)%has_entries () &
        .or. int%children(i)%has_entries ()) then
      write (u, "(1x,A)", advance="no") "internal links:"
      call int%parents(i)%write (u)
      if (int%parents(i)%has_entries ()) &
           write (u, "(1x,A)", advance="no") "=>"
      write (u, "(1x,A)", advance="no") "X"
      if (int%children(i)%has_entries ()) &
           write (u, "(1x,A)", advance="no") "=>"
      call int%children(i)%write (u)
      write (u, *)
   end if
   if (allocated (int%hel_lock)) then
      if (int%hel_lock(i) /= 0) then
         write (u, "(1x,A,1x,I0)")  "helicity lock:", int%hel_lock(i)
      end if
   end if
   if (external_link_is_set (int%source(i))) then
      write (u, "(1x,A)", advance="no") "source:"
      int_link => external_link_get_ptr (int%source(i))
      index_link = external_link_get_index (int%source(i))
      write (u, "(1x,'(',I0,')',I0)", advance="no") &
```

```
                int_link%tag, index_link
            write (u, *)
          end if
       end do
       if (present (show_momentum_sum)) then
          if (allocated (int%p) .and. show_momentum_sum) then
             write (u, "(1x,A)") "Incoming particles (sum):"
             call vector4_write &
                  (sum (int%p(1:int%n_in)), u, show_mass = show_mass)
             write (u, "(1x,A)") "Outgoing particles (sum):"
             call vector4_write &
                  (sum (int%p(int%n_in+int%n_vir+1:)), u, show_mass = show_mass)
             write (u, *)
          end if
       end if
       if (show_st) then
          call state_matrix_write (int%state_matrix, &
               write_value_list=verbose, verbose=verbose, &
               unit=unit, testflag = testflag)
       end if
    else
       write (u, "(1x,A)") "Interaction: [empty]"
    end if
  end subroutine interaction_write
```

Assignment: We implement this as a deep copy. This applies, in particular, to the state-matrix and internal-link components. Furthermore, the new interaction acquires a new tag.

⟨*Interactions: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Interactions: interfaces*⟩≡
```
    interface assignment(=)
       module procedure interaction_assign
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_assign (int_out, int_in)
      type(interaction_t), intent(out) :: int_out
      type(interaction_t), intent(in), target :: int_in
      call interaction_set_tag (int_out)
      int_out%state_matrix = int_in%state_matrix
      int_out%n_in  = int_in%n_in
      int_out%n_out = int_in%n_out
      int_out%n_vir = int_in%n_vir
      int_out%n_tot = int_in%n_tot
      if (allocated (int_in%p_is_known)) then
         allocate (int_out%p_is_known (size (int_in%p_is_known)))
         int_out%p_is_known = int_in%p_is_known
      end if
      if (allocated (int_in%p)) then
         allocate (int_out%p (size (int_in%p)))
         int_out%p = int_in%p
```

```
        end if
        if (allocated (int_in%source)) then
            allocate (int_out%source (size (int_in%source)))
            int_out%source = int_in%source
        end if
        if (allocated (int_in%parents)) then
            allocate (int_out%parents (size (int_in%parents)))
            int_out%parents = int_in%parents
        end if
        if (allocated (int_in%children)) then
            allocate (int_out%children (size (int_in%children)))
            int_out%children = int_in%children
        end if
        if (allocated (int_in%resonant)) then
            allocate (int_out%resonant (size (int_in%resonant)))
            int_out%resonant = int_in%resonant
        end if
        if (allocated (int_in%mask)) then
            allocate (int_out%mask (size (int_in%mask)))
            int_out%mask = int_in%mask
        end if
        if (allocated (int_in%hel_lock)) then
            allocate (int_out%hel_lock (size (int_in%hel_lock)))
            int_out%hel_lock = int_in%hel_lock
        end if
        int_out%update_state_matrix = int_in%update_state_matrix
        int_out%update_values = int_in%update_values
    end subroutine interaction_assign
```

7.6.4 Methods inherited from the state matrix member

Until F2003 is standard, we cannot implement inheritance directly. Therefore, we need wrappers for "inherited" methods.

Make a new branch in the state matrix if it does not yet exist. This is not just a wrapper but it introduces the interaction mask: where a quantum number is masked, it is not transferred but set undefined. After this, the value array has to be updated.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_add_state
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_add_state &
        (int, qn, index, value, sum_values, counter_index, me_index)
      type(interaction_t), intent(inout) :: int
      type(quantum_numbers_t), dimension(:), intent(in) :: qn
      integer, intent(in), optional :: index
      complex(default), intent(in), optional :: value
      logical, intent(in), optional :: sum_values
      integer, intent(in), optional :: counter_index
      integer, intent(out), optional :: me_index
      type(quantum_numbers_t), dimension(size(qn)) :: qn_tmp
      qn_tmp = qn
```

```
        call quantum_numbers_undefine (qn_tmp, int%mask)
        call state_matrix_add_state &
              (int%state_matrix, qn_tmp, index, value, sum_values, &
              counter_index, me_index)
        int%update_values = .true.
      end subroutine interaction_add_state
```

Freeze the quantum state: First collapse the quantum state, i.e., remove quantum numbers if any mask has changed, then fix the array of value pointers.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_freeze
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_freeze (int)
      type(interaction_t), intent(inout) :: int
      if (int%update_state_matrix) then
         call state_matrix_collapse (int%state_matrix, int%mask)
         int%update_state_matrix = .false.
         int%update_values = .true.
      end if
      if (int%update_values) then
         call state_matrix_freeze (int%state_matrix)
         int%update_values = .false.
      end if
    end subroutine interaction_freeze
```

Return true if the state matrix is empty.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_is_empty
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_is_empty (int) result (flag)
      logical :: flag
      type(interaction_t), intent(in) :: int
      flag = state_matrix_is_empty (int%state_matrix)
    end function interaction_is_empty
```

Get the number of values stored in the state matrix:

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_n_matrix_elements
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_n_matrix_elements (int) result (n)
      integer :: n
      type(interaction_t), intent(in) :: int
      n = state_matrix_get_n_matrix_elements (int%state_matrix)
    end function interaction_get_n_matrix_elements
```

Get the norm of the state matrix (if the norm has been taken out, otherwise this would be unity).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_norm
```

⟨*Interactions: procedures*⟩+≡
```
function interaction_get_norm (int) result (norm)
  real(default) :: norm
  type(interaction_t), intent(in) :: int
  norm = state_matrix_get_norm (int%state_matrix)
end function interaction_get_norm
```

Get the quantum number array that corresponds to a given index.
⟨*Interactions: public*⟩+≡
```
public :: interaction_get_quantum_numbers
```
⟨*Interactions: procedures*⟩+≡
```
function interaction_get_quantum_numbers (int, i) result (qn)
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  type(interaction_t), intent(in), target :: int
  integer, intent(in) :: i
  allocate (qn (state_matrix_get_depth (int%state_matrix)))
  qn = state_matrix_get_quantum_numbers (int%state_matrix, i)
end function interaction_get_quantum_numbers
```

Get the matrix element that corresponds to a set of quantum numbers, a given index, or return the whole array.
⟨*Interactions: public*⟩+≡
```
public :: interaction_get_matrix_element
```
⟨*Interactions: procedures*⟩+≡
```
function interaction_get_matrix_element (int, i) result (me)
  complex(default) :: me
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  me = state_matrix_get_matrix_element (int%state_matrix, i)
end function interaction_get_matrix_element
```

Set the complex value(s) stored in the quantum state.
⟨*Interactions: public*⟩+≡
```
public :: interaction_set_matrix_element
```
⟨*Interactions: interfaces*⟩+≡
```
interface interaction_set_matrix_element
   module procedure interaction_set_matrix_element_qn
   module procedure interaction_set_matrix_element_all
   module procedure interaction_set_matrix_element_array
   module procedure interaction_set_matrix_element_single
   module procedure interaction_set_matrix_element_clone
end interface
```
Indirect access via the quantum number array:
⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_set_matrix_element_qn (int, qn, val)
  type(interaction_t), intent(inout) :: int
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  complex(default), intent(in) :: val
  call state_matrix_set_matrix_element (int%state_matrix, qn, val)
end subroutine interaction_set_matrix_element_qn
```

799

Set all entries of the matrix-element array to a given value.

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_set_matrix_element_all (int, value)
  type(interaction_t), intent(inout) :: int
  complex(default), intent(in) :: value
  call state_matrix_set_matrix_element (int%state_matrix, value)
end subroutine interaction_set_matrix_element_all
```

Set the matrix-element array directly.

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_set_matrix_element_array (int, value)
  type(interaction_t), intent(inout) :: int
  complex(default), dimension(:), intent(in) :: value
  call state_matrix_set_matrix_element (int%state_matrix, value)
end subroutine interaction_set_matrix_element_array

pure subroutine interaction_set_matrix_element_single (int, i, value)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  complex(default), intent(in) :: value
  call state_matrix_set_matrix_element (int%state_matrix, i, value)
end subroutine interaction_set_matrix_element_single
```

Clone from another (matching) interaction.

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_set_matrix_element_clone (int, int1)
  type(interaction_t), intent(inout) :: int
  type(interaction_t), intent(in) :: int1
  call state_matrix_set_matrix_element (int%state_matrix, int1%state_matrix)
end subroutine interaction_set_matrix_element_clone
```

Get the indices of any diagonal matrix elements.

⟨*Interactions: public*⟩+≡
```
public :: interaction_get_diagonal_entries
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_get_diagonal_entries (int, i)
  type(interaction_t), intent(in) :: int
  integer, dimension(:), allocatable, intent(out) :: i
  call state_matrix_get_diagonal_entries (int%state_matrix, i)
end subroutine interaction_get_diagonal_entries
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

⟨*Interactions: public*⟩+≡
```
public :: interaction_normalize_by_trace
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_normalize_by_trace (int)
  type(interaction_t), intent(inout) :: int
  call state_matrix_normalize_by_trace (int%state_matrix)
end subroutine interaction_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_normalize_by_max
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_normalize_by_max (int)
      type(interaction_t), intent(inout) :: int
      call state_matrix_normalize_by_max (int%state_matrix)
    end subroutine interaction_normalize_by_max
```

Explicitly set the norm value (of the state matrix).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_norm
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_set_norm (int, norm)
      type(interaction_t), intent(inout) :: int
      real(default), intent(in) :: norm
      call state_matrix_set_norm (int%state_matrix, norm)
    end subroutine interaction_set_norm
```

Return the maximum absolute value of color indices.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_max_color_value
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_max_color_value (int) result (cmax)
      integer :: cmax
      type(interaction_t), intent(in) :: int
      cmax = state_matrix_get_max_color_value (int%state_matrix)
    end function interaction_get_max_color_value
```

Factorize the state matrix into single-particle state matrices, the branch selection depending on a (random) value between 0 and 1; optionally also return a correlated state matrix.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_factorize
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_factorize &
        (int, mode, x, ok, single_state, correlated_state, qn_in)
      type(interaction_t), intent(in), target :: int
      integer, intent(in) :: mode
      real(default), intent(in) :: x
      logical, intent(out) :: ok
      type(state_matrix_t), &
          dimension(:), allocatable, intent(out) :: single_state
      type(state_matrix_t), intent(out), optional :: correlated_state
      type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
      call state_matrix_factorize &
          (int%state_matrix, mode, x, ok, single_state, correlated_state, qn_in)
    end subroutine interaction_factorize
```

Sum all matrix element values

⟨*Interactions: public*⟩+≡
```
    public :: interaction_sum
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_sum (int) result (value)
      complex(default) :: value
      type(interaction_t), intent(in) :: int
      value = state_matrix_sum (int%state_matrix)
    end function interaction_sum
```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no **freeze** must be performed anymore.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_add_color_contractions
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_add_color_contractions (int)
      type(interaction_t), intent(inout) :: int
      call state_matrix_add_color_contractions (int%state_matrix)
    end subroutine interaction_add_color_contractions
```

Multiply matrix elements from two interactions. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_evaluate_product
    public :: interaction_evaluate_product_cf
    public :: interaction_evaluate_square_c
    public :: interaction_evaluate_sum
    public :: interaction_evaluate_me_sum
```

⟨*Interactions: procedures*⟩+≡
```
    pure subroutine interaction_evaluate_product &
        (int, i, int1, int2, index1, index2)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in) :: int1, int2
      integer, dimension(:), intent(in) :: index1, index2
      call state_matrix_evaluate_product &
          (int%state_matrix, i, int1%state_matrix, int2%state_matrix, &
           index1, index2)
    end subroutine interaction_evaluate_product

    pure subroutine interaction_evaluate_product_cf &
        (int, i, int1, int2, index1, index2, factor)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in) :: int1, int2
      integer, dimension(:), intent(in) :: index1, index2
      complex(default), dimension(:), intent(in) :: factor
```

```
      call state_matrix_evaluate_product_cf &
           (int%state_matrix, i, int1%state_matrix, int2%state_matrix, &
            index1, index2, factor)
    end subroutine interaction_evaluate_product_cf

    pure subroutine interaction_evaluate_square_c (int, i, int1, index1)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in) :: int1
      integer, dimension(:), intent(in) :: index1
      call state_matrix_evaluate_square_c &
           (int%state_matrix, i, int1%state_matrix, index1)
    end subroutine interaction_evaluate_square_c

    pure subroutine interaction_evaluate_sum (int, i, int1, index1)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in) :: int1
      integer, dimension(:), intent(in) :: index1
      call state_matrix_evaluate_sum &
           (int%state_matrix, i, int1%state_matrix, index1)
    end subroutine interaction_evaluate_sum

    pure subroutine interaction_evaluate_me_sum (int, i, int1, index1)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in) :: int1
      integer, dimension(:), intent(in) :: index1
      call state_matrix_evaluate_me_sum &
           (int%state_matrix, i, int1%state_matrix, index1)
    end subroutine interaction_evaluate_me_sum
```

7.6.5 Accessing contents

Return the integer tag.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_tag
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_tag (int) result (tag)
      integer :: tag
      type(interaction_t), intent(in) :: int
      tag = int%tag
    end function interaction_get_tag
```

Return the number of particles.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_n_tot
    public :: interaction_get_n_in
    public :: interaction_get_n_vir
    public :: interaction_get_n_out
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_n_tot (int) result (n_tot)
  integer :: n_tot
  type(interaction_t), intent(in) :: int
  n_tot = int%n_tot
end function interaction_get_n_tot

function interaction_get_n_in (int) result (n_in)
  integer :: n_in
  type(interaction_t), intent(in) :: int
  n_in = int%n_in
end function interaction_get_n_in

function interaction_get_n_vir (int) result (n_vir)
  integer :: n_vir
  type(interaction_t), intent(in) :: int
  n_vir = int%n_vir
end function interaction_get_n_vir

function interaction_get_n_out (int) result (n_out)
  integer :: n_out
  type(interaction_t), intent(in) :: int
  n_out = int%n_out
end function interaction_get_n_out
```

Return a momentum index. The flags specify whether to keep/drop incoming, virtual, or outgoing momenta. Check for illegal values.

⟨*Interactions: procedures*⟩+≡

```
function idx (int, i, outgoing)
  integer :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  logical, intent(in), optional :: outgoing
  logical :: in, vir, out
  if (present (outgoing)) then
     in  = .not. outgoing
     vir = .false.
     out = outgoing
  else
     in = .true.
     vir = .true.
     out = .true.
  end if
  idx = 0
  if (in) then
     if (vir) then
        if (out) then
           if (i <= int%n_tot)  idx = i
        else
           if (i <= int%n_in + int%n_vir)  idx = i
        end if
     else if (out) then
        if (i <= int%n_in) then
```

804

```
            idx = i
         else if (i <= int%n_in + int%n_out) then
            idx = int%n_vir + i
         end if
      else
         if (i <= int%n_in)  idx = i
      end if
   else if (vir) then
      if (out) then
         if (i <= int%n_vir + int%n_out)  idx = int%n_in + i
      else
         if (i <= int%n_vir)  idx = int%n_in + i
      end if
   else if (out) then
      if (i <= int%n_out)  idx = int%n_in + int%n_vir + i
   end if
   if (idx == 0) then
      call interaction_write (int)
      print *, i, in, vir, out
      call msg_bug (" Momentum index is out of range for this interaction")
   end if
end function idx
```

Return all or just a specific four-momentum.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_momenta
    public :: interaction_get_momentum
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_get_momenta
      module procedure interaction_get_momenta_all
      module procedure interaction_get_momenta_idx
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_momenta_all (int, outgoing) result (p)
      type(vector4_t), dimension(:), allocatable :: p
      type(interaction_t), intent(in) :: int
      logical, intent(in), optional :: outgoing
      integer :: i
      if (present (outgoing)) then
         if (outgoing) then
            allocate (p (int%n_out))
         else
            allocate (p (int%n_in))
         end if
      else
         allocate (p (int%n_tot))
      end if
      do i = 1, size (p)
         p(i) = int%p(idx (int, i, outgoing))
      end do
    end function interaction_get_momenta_all

    function interaction_get_momenta_idx (int, jj) result (p)
```

```
      type(vector4_t), dimension(:), allocatable :: p
      type(interaction_t), intent(in) :: int
      integer, dimension(:), intent(in) :: jj
      allocate (p (size (jj)))
      p = int%p(jj)
    end function interaction_get_momenta_idx

    function interaction_get_momentum (int, i, outgoing) result (p)
      type(vector4_t) :: p
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      logical, intent(in), optional :: outgoing
      p = int%p(idx (int, i, outgoing))
    end function interaction_get_momentum
```

This is a variant as a subroutine. Redundant, but the function above fails at times for gfortran 4.5.0 (double allocation, compiler bug).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_momenta_sub
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_get_momenta_sub (int, p, outgoing)
      type(vector4_t), dimension(:), intent(out) :: p
      type(interaction_t), intent(in) :: int
      logical, intent(in), optional :: outgoing
      integer :: i
      do i = 1, size (p)
         p(i) = int%p(idx (int, i, outgoing))
      end do
    end subroutine interaction_get_momenta_sub
```

Transfer PDG codes, masses (initalization) and momenta to a predefined subevent. We use the flavor assignment of the first branch in the interaction state matrix. Only incoming and outgoing particles are transferred. Switch momentum sign for incoming particles.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_to_subevt
    public :: interaction_momenta_to_subevt
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_momenta_to_subevt
       module procedure interaction_momenta_to_subevt_id
       module procedure interaction_momenta_to_subevt_tr
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_to_subevt (int, j_beam, j_in, j_out, subevt)
      type(interaction_t), intent(in), target :: int
      integer, dimension(:), intent(in) :: j_beam, j_in, j_out
      type(subevt_t), intent(out) :: subevt
      type(flavor_t), dimension(:), allocatable :: flv
      integer :: n_beam, n_in, n_out, i, j
      allocate (flv (int%n_tot))
```

```
      flv = quantum_numbers_get_flavor (interaction_get_quantum_numbers (int, 1))
      n_beam = size (j_beam)
      n_in = size (j_in)
      n_out = size (j_out)
      call subevt_init (subevt, n_beam + n_in + n_out)
      do i = 1, n_beam
         j = j_beam(i)
         call subevt_set_beam (subevt, i, &
              flavor_get_pdg (flv(j)), &
              vector4_null, &
              flavor_get_mass (flv(j)) ** 2)
      end do
      do i = 1, n_in
         j = j_in(i)
         call subevt_set_incoming (subevt, n_beam + i, &
              flavor_get_pdg (flv(j)), &
              vector4_null, &
              flavor_get_mass (flv(j)) ** 2)
      end do
      do i = 1, n_out
         j = j_out(i)
         call subevt_set_outgoing (subevt, n_beam + n_in + i, &
              flavor_get_pdg (flv(j)), &
              vector4_null, &
              flavor_get_mass (flv(j)) ** 2)
      end do
    end subroutine interaction_to_subevt

    subroutine interaction_momenta_to_subevt_id (int, j_beam, j_in, j_out, subevt)
      type(interaction_t), intent(in) :: int
      integer, dimension(:), intent(in) :: j_beam, j_in, j_out
      type(subevt_t), intent(inout) :: subevt
      call subevt_set_p_beam &
           (subevt, - interaction_get_momenta (int, j_beam))
      call subevt_set_p_incoming &
           (subevt, - interaction_get_momenta (int, j_in))
      call subevt_set_p_outgoing &
           (subevt, interaction_get_momenta (int, j_out))
    end subroutine interaction_momenta_to_subevt_id

    subroutine interaction_momenta_to_subevt_tr &
         (int, j_beam, j_in, j_out, lt, subevt)
      type(interaction_t), intent(in) :: int
      integer, dimension(:), intent(in) :: j_beam, j_in, j_out
      type(subevt_t), intent(inout) :: subevt
      type(lorentz_transformation_t), intent(in) :: lt
      call subevt_set_p_beam &
           (subevt, - lt * interaction_get_momenta (int, j_beam))
      call subevt_set_p_incoming &
           (subevt, - lt * interaction_get_momenta (int, j_in))
      call subevt_set_p_outgoing &
           (subevt, lt * interaction_get_momenta (int, j_out))
    end subroutine interaction_momenta_to_subevt_tr
```

Return a shallow copy of the state matrix:

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_state_matrix_ptr
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_state_matrix_ptr (int) result (state)
      type(state_matrix_t), pointer :: state
      type(interaction_t), intent(in), target :: int
      state => int%state_matrix
    end function interaction_get_state_matrix_ptr
```

Return the array of resonance flags

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_resonance_flags
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_resonance_flags (int) result (resonant)
      type(interaction_t), intent(in) :: int
      logical, dimension(size(int%resonant)) :: resonant
      resonant = int%resonant
    end function interaction_get_resonance_flags
```

Return the quantum-numbers mask (or part of it)

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_mask
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_get_mask
       module procedure interaction_get_mask_all
       module procedure interaction_get_mask_slice
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_mask_all (int) result (mask)
      type(interaction_t), intent(in) :: int
      type(quantum_numbers_mask_t), dimension(size(int%mask)) :: mask
      mask = int%mask
    end function interaction_get_mask_all

    function interaction_get_mask_slice (int, index) result (mask)
      type(interaction_t), intent(in) :: int
      integer, dimension(:), intent(in) :: index
      type(quantum_numbers_mask_t), dimension(size(index)) :: mask
      mask = int%mask(index)
    end function interaction_get_mask_slice
```

Compute the invariant mass squared of the incoming particles (if any, otherwise outgoing).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_s
```

⟨*Interactions: procedures*⟩+≡
```
function interaction_get_s (int) result (s)
  real(default) :: s
  type(interaction_t), intent(in) :: int
  if (int%n_in /= 0) then
     s = sum (int%p(:int%n_in)) ** 2
  else
     s = sum (int%p(int%n_vir+1:)) ** 2
  end if
end function interaction_get_s
```

Compute the Lorentz transformation that transforms the incoming particles from the center-of-mass frame to the lab frame where they are given. If the c.m. mass squared is negative or zero, return the identity.

⟨*Interactions: public*⟩+≡
```
public :: interaction_get_cm_transformation
```

⟨*Interactions: procedures*⟩+≡
```
function interaction_get_cm_transformation (int) result (lt)
  type(lorentz_transformation_t) :: lt
  type(interaction_t), intent(in) :: int
  type(vector4_t) :: p_cm
  real(default) :: s
  if (int%n_in /= 0) then
     p_cm = sum (int%p(:int%n_in))
  else
     p_cm = sum (int%p(int%n_vir+1:))
  end if
  s = p_cm ** 2
  if (s > 0) then
     lt = boost (p_cm, sqrt (s))
  else
     lt = identity
  end if
end function interaction_get_cm_transformation
```

Return flavor, momentum, and position of the first outgoing unstable particle present in the interaction. Note that we need not iterate through the state matrix; if there is an unstable particle, it will be present in all state-matrix entries.

⟨*Interactions: public*⟩+≡
```
public :: interaction_get_unstable_particle
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_get_unstable_particle (int, flv, p, i)
  type(interaction_t), intent(in), target :: int
  type(flavor_t), intent(out) :: flv
  type(vector4_t), intent(out) :: p
  integer, intent(out) :: i
  type(state_iterator_t) :: it
  type(flavor_t), dimension(int%n_tot) :: flv_array
  call state_iterator_init (it, int%state_matrix)
  flv_array = state_iterator_get_flavor (it)
```

```
        do i = int%n_in + int%n_vir + 1, int%n_tot
          if (.not. flavor_is_stable (flv_array(i))) then
            flv = flv_array(i)
            p = int%p(i)
            return
          end if
        end do
      end subroutine interaction_get_unstable_particle
```

Return the complete set of *outgoing* flavors, assuming that the flavor quantum number is not suppressed.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_flv_out
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_get_flv_out (int, flv)
      type(interaction_t), intent(in), target :: int
      type(flavor_t), dimension(:,:), allocatable, intent(out) :: flv
      type(state_iterator_t) :: it
      type(flavor_t), dimension(:), allocatable :: flv_state
      integer :: n_in, n_vir, n_out, n_tot, n_state, i
      n_in = interaction_get_n_in (int)
      n_vir = interaction_get_n_vir (int)
      n_out = interaction_get_n_out (int)
      n_tot = interaction_get_n_tot (int)
      n_state = interaction_get_n_matrix_elements (int)
      allocate (flv (n_out, n_state))
      allocate (flv_state (n_tot))
      i = 1
      call state_iterator_init (it, interaction_get_state_matrix_ptr (int))
      do while (state_iterator_is_valid (it))
        flv_state = state_iterator_get_flavor (it)
        flv(:,i) = flv_state(n_in+n_vir+1:)
        i = i + 1
        call state_iterator_advance (it)
      end do
    end subroutine interaction_get_flv_out
```

7.6.6 Modifying contents

Set the quantum numbers mask.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_mask
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_set_mask (int, mask)
      type(interaction_t), intent(inout) :: int
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      int%mask = mask
      int%update_state_matrix = .true.
    end subroutine interaction_set_mask
```

Merge a particular mask entry, respecting a possible helicity lock for this entry. We apply an OR relation, which means that quantum numbers are summed over if either of the two masks requires it.

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_merge_mask_entry (int, i, mask)
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(quantum_numbers_mask_t), intent(in) :: mask
  type(quantum_numbers_mask_t) :: mask_tmp
  integer :: ii
  ii = idx (int, i)
  if (int%mask(ii) .neqv. mask) then
     int%mask(ii) = int%mask(ii) .or. mask
     if (int%hel_lock(ii) /= 0) then
        call quantum_numbers_mask_assign (mask_tmp, mask, helicity=.true.)
        int%mask(int%hel_lock(ii)) = int%mask(int%hel_lock(ii)) .or. mask_tmp
     end if
  end if
  int%update_state_matrix = .true.
end subroutine interaction_merge_mask_entry
```

Fill the momenta array, do not care about the quantum numbers of particles.

⟨*Interactions: public*⟩+≡

```
public :: interaction_reset_momenta
public :: interaction_set_momenta
public :: interaction_set_momentum
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_reset_momenta (int)
  type(interaction_t), intent(inout) :: int
  int%p = vector4_null
  int%p_is_known = .true.
end subroutine interaction_reset_momenta

subroutine interaction_set_momenta (int, p, outgoing)
  type(interaction_t), intent(inout) :: int
  type(vector4_t), dimension(:), intent(in) :: p
  logical, intent(in), optional :: outgoing
  integer :: i, index
  do i = 1, size (p)
     index = idx (int, i, outgoing)
     int%p(index) = p(i)
     int%p_is_known(index) = .true.
  end do
end subroutine interaction_set_momenta

subroutine interaction_set_momentum (int, p, i, outgoing)
  type(interaction_t), intent(inout) :: int
  type(vector4_t), intent(in) :: p
  integer, intent(in) :: i
  logical, intent(in), optional :: outgoing
  integer :: index
  index = idx (int, i, outgoing)
  int%p(index) = p
```

```
        int%p_is_known(index) = .true.
      end subroutine interaction_set_momentum
```

This more sophisticated version of setting values is used for structure functions, in particular if nontrivial flavor, color, and helicity may be present: set values selectively for the given flavors. If there is more than one flavor, scan the interaction and check for a matching flavor at the specified particle location. If it matches, insert the value that corresponds to this flavor.

⟨*Interactions: public*⟩+≡
```
      public :: interaction_set_flavored_values
```

⟨*Interactions: procedures*⟩+≡
```
      subroutine interaction_set_flavored_values (int, value, flv_in, pos)
        type(interaction_t), intent(inout) :: int
        complex(default), dimension(:), intent(in) :: value
        type(flavor_t), dimension(:), intent(in) :: flv_in
        integer, intent(in) :: pos
        type(state_iterator_t) :: it
        type(flavor_t) :: flv
        integer :: i
        if (size (value) == 1) then
           call interaction_set_matrix_element (int, value(1))
        else
           call state_iterator_init (it, int%state_matrix)
           do while (state_iterator_is_valid (it))
              flv = state_iterator_get_flavor (it, pos)
              SCAN_FLV: do i = 1, size (value)
                 if (flv == flv_in(i)) then
                    call state_iterator_set_matrix_element (it, value(i))
                    exit SCAN_FLV
                 end if
              end do SCAN_FLV
              call state_iterator_advance (it)
           end do
        end if
      end subroutine interaction_set_flavored_values
```

7.6.7   Handling Linked interactions

Store relations between corresponding particles within one interaction. The first particle is the parent, the second one the child. Links are established in both directions.

These relations have no effect on the propagation of momenta etc., they are rather used for mother-daughter relations in event output.

⟨*Interactions: public*⟩+≡
```
      public :: interaction_relate
```

⟨*Interactions: procedures*⟩+≡
```
      subroutine interaction_relate (int, i1, i2)
        type(interaction_t), intent(inout), target :: int
        integer, intent(in) :: i1, i2
        if (i1 /= 0 .and. i2 /= 0) then
```

```
      call int%children(i1)%append (i2)
      call int%parents(i2)%append (i1)
    end if
  end subroutine interaction_relate
```

Transfer internal parent-child relations defined within interaction `int1` to a new interaction `int` where the particle indices are mapped to. Some particles in `int1` may have no image in `int`. In that case, a child entry maps to zero, and we skip this relation.

Also transfer resonance flags.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_transfer_relations
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_transfer_relations (int1, int2, map)
      type(interaction_t), intent(in) :: int1
      type(interaction_t), intent(inout), target :: int2
      integer, dimension(:), intent(in) :: map
      integer :: i, j, k
      do i = 1, size (map)
        do j = 1, int1%parents(i)%get_length ()
          k = int1%parents(i)%get_link (j)
          call interaction_relate (int2, map(k), map(i))
        end do
        if (map(i) /= 0) then
          int2%resonant(map(i)) = int1%resonant(i)
        end if
      end do
    end subroutine interaction_transfer_relations
```

Make up internal parent-child relations for the particle(s) that are connected to a new interaction `int`.

If `resonant` is defined and true, the connections are marked as resonant in the result interaction

⟨*Interactions: public*⟩+≡
```
    public :: interaction_relate_connections
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_relate_connections &
        (int, int_in, connection_index, &
         map, map_connections, resonant)
      type(interaction_t), intent(inout), target :: int
      type(interaction_t), intent(in) :: int_in
      integer, dimension(:), intent(in) :: connection_index
      integer, dimension(:), intent(in) :: map, map_connections
      logical, intent(in), optional :: resonant
      logical :: reson
      integer :: i, j, i2, k2
      reson = .false.;  if (present (resonant))  reson = resonant
      do i = 1, size (map_connections)
        k2 = connection_index(i)
        do j = 1, int_in%children(k2)%get_length ()
          i2 = int_in%children(k2)%get_link (j)
```

```
        call interaction_relate (int, map_connections(i), map(i2))
      end do
      int%resonant(map_connections(i)) = reson
    end do
  end subroutine interaction_relate_connections
```

Return the number of source/target links of the internal connections of particle
i.
⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_n_children
    public :: interaction_get_n_parents
```
⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_n_children (int, i) result (n)
      integer :: n
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      n = int%children(i)%get_length ()
    end function interaction_get_n_children

    function interaction_get_n_parents (int, i) result (n)
      integer :: n
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      n = int%parents(i)%get_length ()
    end function interaction_get_n_parents
```

Return the source/target links of the internal connections of particle i as an
array.
⟨*Interactions: public*⟩+≡
```
    public :: interaction_get_children
    public :: interaction_get_parents
```
⟨*Interactions: procedures*⟩+≡
```
    function interaction_get_children (int, i) result (idx)
      integer, dimension(:), allocatable :: idx
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      integer :: k, l
      l = int%children(i)%get_length ()
      allocate (idx (l))
      do k = 1, l
          idx(k) = int%children(i)%get_link (k)
      end do
    end function interaction_get_children

    function interaction_get_parents (int, i) result (idx)
      integer, dimension(:), allocatable :: idx
      type(interaction_t), intent(in) :: int
      integer, intent(in) :: i
      integer :: k, l
      l = int%parents(i)%get_length ()
      allocate (idx (l))
      do k = 1, l
```

```
        idx(k) = int%parents(i)%get_link (k)
      end do
    end function interaction_get_parents
```

Add a source link from an interaction to a corresponding particle within another
interaction. These links affect the propagation of particles: the two linked
particles are considered as the same particle, outgoing and incoming.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_set_source_link
```

⟨*Interactions: interfaces*⟩+≡
```
    interface interaction_set_source_link
       module procedure interaction_set_source_link_int
    end interface
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_set_source_link_int (int, i, int1, i1)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: i
      type(interaction_t), intent(in), target :: int1
      integer, intent(in) :: i1
      if (i /= 0)  call external_link_set (int%source(i), int1, i1)
    end subroutine interaction_set_source_link_int
```

Reassign links to a new interaction (which is an image of the current interaction).

⟨*Interactions: public*⟩+≡
```
    public :: interaction_reassign_links
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_reassign_links (int, int_src, int_target)
      type(interaction_t), intent(inout) :: int
      type(interaction_t), intent(in) :: int_src
      type(interaction_t), intent(in), target :: int_target
      integer :: i
      if (allocated (int%source)) then
        do i = 1, size (int%source)
           call external_link_reassign (int%source(i), int_src, int_target)
        end do
      end if
    end subroutine interaction_reassign_links
```

Since links are one-directional, if we want to follow them backwards we have
to scan all possibilities. This procedure returns the index of the particle within
int which points to the particle i1 within interaction int1. If unsuccessful,
return zero.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_find_link
```

⟨*Interactions: procedures*⟩+≡
```
    function interaction_find_link (int, int1, i1) result (i)
      integer :: i
      type(interaction_t), intent(in) :: int, int1
      integer, intent(in) :: i1
      type(interaction_t), pointer :: int_tmp
```

```
        do i = 1, int%n_tot
           int_tmp => external_link_get_ptr (int%source(i))
           if (int_tmp%tag == int1%tag) then
              if (external_link_get_index (int%source(i)) == i1)  return
           end if
        end do
        i = 0
     end function interaction_find_link
```

The inverse: return interaction pointer and index for the ultimate source of i
within int.

⟨*Interactions: public*⟩+≡
```
     public :: interaction_find_source
```

⟨*Interactions: procedures*⟩+≡
```
     subroutine interaction_find_source (int, i, int1, i1)
       type(interaction_t), intent(in) :: int
       integer, intent(in) :: i
       type(interaction_t), intent(out), pointer :: int1
       integer, intent(out) :: i1
       type(external_link_t) :: link
       link = interaction_get_ultimate_source (int, i)
       int1 => external_link_get_ptr (link)
       i1 = external_link_get_index (link)
     end subroutine interaction_find_source
```

Follow source links recursively to return the ultimate source of a particle.

⟨*Interactions: procedures*⟩+≡
```
     function interaction_get_ultimate_source (int, i) result (link)
       type(external_link_t) :: link
       type(interaction_t), intent(in) :: int
       integer, intent(in) :: i
       type(interaction_t), pointer :: int_src
       integer :: i_src
       link = int%source(i)
       if (external_link_is_set (link)) then
          do
             int_src => external_link_get_ptr (link)
             i_src = external_link_get_index (link)
             if (external_link_is_set (int_src%source(i_src))) then
                link = int_src%source(i_src)
             else
                exit
             end if
          end do
       end if
     end function interaction_get_ultimate_source
```

Update mask entries by merging them with corresponding masks in interactions
linked to the current one. The mask determines quantum numbers which are
summed over.

Note that both the mask of the current interaction and the mask of the

linked interaction are updated (side effect!). This ensures that both agree for
the linked particle.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_exchange_mask
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_exchange_mask (int)
      type(interaction_t), intent(inout) :: int
      integer :: i, index_link
      type(interaction_t), pointer :: int_link
      do i = 1, int%n_tot
         if (external_link_is_set (int%source(i))) then
            int_link => external_link_get_ptr (int%source(i))
            index_link = external_link_get_index (int%source(i))
            call interaction_merge_mask_entry &
                 (int, i, int_link%mask(index_link))
            call interaction_merge_mask_entry &
                 (int_link, index_link, int%mask(i))
         end if
      end do
      call interaction_freeze (int)
    end subroutine interaction_exchange_mask
```

Copy momenta from interactions linked to the current one.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_receive_momenta
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_receive_momenta (int)
      type(interaction_t), intent(inout) :: int
      integer :: i, index_link
      type(interaction_t), pointer :: int_link
      do i = 1, int%n_tot
         if (external_link_is_set (int%source(i))) then
            int_link => external_link_get_ptr (int%source(i))
            index_link = external_link_get_index (int%source(i))
            call interaction_set_momentum (int, int_link%p(index_link), i)
         end if
      end do
    end subroutine interaction_receive_momenta
```

The inverse operation: Copy momenta back to the interactions linked to the
current one.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_send_momenta
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_send_momenta (int)
      type(interaction_t), intent(in) :: int
      integer :: i, index_link
      type(interaction_t), pointer :: int_link
      do i = 1, int%n_tot
         if (external_link_is_set (int%source(i))) then
            int_link => external_link_get_ptr (int%source(i))
```

```
                index_link = external_link_get_index (int%source(i))
                call interaction_set_momentum (int_link, int%p(i), index_link)
            end if
        end do
    end subroutine interaction_send_momenta
```

For numerical comparisons: pacify all momenta in an interaction.

⟨*Interactions: public*⟩+≡
```
    public :: interaction_pacify_momenta
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_pacify_momenta (int, acc)
      type(interaction_t), intent(inout) :: int
      real(default), intent(in) :: acc
      integer :: i
      do i = 1, int%n_tot
         call pacify (int%p(i), acc)
      end do
    end subroutine interaction_pacify_momenta
```

7.6.8   Recovering connections

When creating an evaluator for two interactions, we have to know by which
particles they are connected. The connection indices can be determined if we
have two linked interactions. We assume that `int1` is the source and `int2` the
target, so the connections of interest are stored within `int2`. A connection is
found if either the source is `int1`, or the (ultimate) source of a particle within
`int2` coincides with the (ultimate) source of a aparticle within `int1`. The result
is an array of index pairs.

To make things simple, we scan the interaction twice, once for counting hits,
then allocate the array, then scan again and store the connections.

The connections are scanned for `int2`, which has sources in `int1`. It may
happen that the order of connections is interchanged (crossed). We require the
indices in `int1` to be sorted, so we reorder both index arrays correspondingly
before returning them. (After this, the indices in `int2` may be out of order.)

⟨*Interactions: public*⟩+≡
```
    public :: find_connections
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine find_connections (int1, int2, n, connection_index)
      type(interaction_t), intent(in) :: int1, int2
      integer, intent(out) :: n
      integer, dimension(:,:), intent(out), allocatable :: connection_index
      integer, dimension(:,:), allocatable :: conn_index_tmp
      integer, dimension(:), allocatable :: ordering
      integer :: i, j, k
      type(external_link_t) :: link2, link1
      type(interaction_t), pointer :: int_link, int_link1
      n = 0
      do i = 1, size (int2%source)
         link2 = interaction_get_ultimate_source (int2, i)
```

```
    if (external_link_is_set (link2)) then
        int_link => external_link_get_ptr (link2)
        if (int_link%tag == int1%tag) then
            n = n + 1
        else
            k = external_link_get_index (link2)
            do j = 1, size (int1%source)
                link1 = interaction_get_ultimate_source (int1, j)
                if (external_link_is_set (link1)) then
                    int_link1 => external_link_get_ptr (link1)
                    if (int_link1%tag == int_link%tag) then
                        if (external_link_get_index (link1) == k) then
                            n = n + 1
                        end if
                    end if
                end if
            end do
        end if
    end if
end do
allocate (conn_index_tmp (n, 2))
n = 0
do i = 1, size (int2%source)
    link2 = interaction_get_ultimate_source (int2, i)
    if (external_link_is_set (link2)) then
        int_link => external_link_get_ptr (link2)
        if (int_link%tag == int1%tag) then
            n = n + 1
            conn_index_tmp(n,1) = external_link_get_index (int2%source(i))
            conn_index_tmp(n,2) = i
        else
            k = external_link_get_index (link2)
            do j = 1, size (int1%source)
                link1 = interaction_get_ultimate_source (int1, j)
                if (external_link_is_set (link1)) then
                    int_link1 => external_link_get_ptr (link1)
                    if (int_link1%tag == int_link%tag) then
                        if (external_link_get_index (link1) == k) then
                            n = n + 1
                            conn_index_tmp(n,1) = j
                            conn_index_tmp(n,2) = i
                        end if
                    end if
                end if
            end do
        end if
    end if
end do
allocate (connection_index (n, 2))
if (n > 1) then
    allocate (ordering (n))
    ordering = order (conn_index_tmp(:,1))
    connection_index = conn_index_tmp(ordering,:)
else
```

```
          connection_index = conn_index_tmp
        end if
    end subroutine find_connections
```

7.6.9   Unit tests

⟨*Interactions: public*⟩+≡
```
    public :: interaction_test
```

⟨*Interactions: procedures*⟩+≡
```
    subroutine interaction_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Interactions: execute tests⟩
    end subroutine interaction_test
```

Generate an interaction of a polarized virtual photon and a colored quark which
may be either up or down. Remove the quark polarization. Generate another
interaction for the quark radiating a photon and link this to the first interation.
The radiation ignores polarization; transfer this information to the first inter-
action to simplify it. Then, transfer the momentum to the radiating quark and
perform a splitting.

⟨*Interactions: execute tests*⟩≡
```
    call test (interaction_1, "interaction_1", &
         "check interaction setup", &
         u, results)
```

⟨*Interactions: tests*⟩≡
```
    subroutine interaction_1 (u)
      integer, intent(in) :: u
      type(interaction_t), target :: int, rad
      type(vector4_t), dimension(3) :: p
      type(quantum_numbers_mask_t), dimension(3) :: mask
      p(2) = vector4_moving (500._default, 500._default, 1)
      p(3) = vector4_moving (500._default,-500._default, 1)
      p(1) = p(2) + p(3)

      write (u, "(A)")  "* Test output: interaction"
      write (u, "(A)")  "*   Purpose: check routines for interactions"
      write (u, "(A)")

      call interaction_init (int, 1, 0, 2, set_relations=.true., &
           store_values = .true. )
      call int_set (int, 1, -1, 1, 1, &
           cmplx (0.3_default, 0.1_default, kind=default))
      call int_set (int, 1, -1,-1, 1, &
           cmplx (0.5_default,-0.7_default, kind=default))
      call int_set (int, 1, 1, 1, 1, &
           cmplx (0.1_default, 0._default, kind=default))
      call int_set (int, -1, 1, -1, 2, &
           cmplx (0.4_default, -0.1_default, kind=default))
      call int_set (int, 1, 1, 1, 2, &
           cmplx (0.2_default, 0._default, kind=default))
```

```fortran
    call interaction_freeze (int)
    call interaction_set_momenta (int, p)
  mask = new_quantum_numbers_mask (.false.,.false., (/.true.,.true.,.true./))
    call interaction_init (rad, 1, 0, 2, &
         mask=mask, set_relations=.true., store_values = .true.)
    call rad_set (1)
    call rad_set (2)
    call interaction_set_source_link (rad, 1, int, 2)
    call interaction_exchange_mask (rad)
    call interaction_receive_momenta (rad)
    p(1) = interaction_get_momentum (rad, 1)
    p(2) = 0.4_default * p(1)
    p(3) = p(1) - p(2)
    call interaction_set_momenta (rad, p(2:3), outgoing=.true.)
    call interaction_freeze (int)
    call interaction_freeze (rad)
    call interaction_set_matrix_element &
         (rad, cmplx (0._default, 0._default, kind=default))
    call interaction_write (int, u)
    write (u, "(A)")
    call interaction_write (rad, u)
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"
    call interaction_final (int)
    call interaction_final (rad)
    write (u, "(A)")
    write (u, "(A)")  "* Test interaction_1: successful."
contains
  subroutine int_set (int, h1, h2, hq, q, val)
    type(interaction_t), target, intent(inout) :: int
    integer, intent(in) :: h1, h2, hq, q
    type(flavor_t), dimension(3) :: flv
    type(color_t), dimension(3) :: col
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    complex(default), intent(in) :: val
    call flavor_init (flv, (/21, q, -q/))
    call color_init_col_acl (col(2), 5, 0)
    call color_init_col_acl (col(3), 0, 5)
    call helicity_init (hel, [h1, hq, -hq], [h2, hq, -hq])
    call quantum_numbers_init (qn, flv, col, hel)
    call interaction_add_state (int, qn)
    call interaction_set_matrix_element (int, val)
  end subroutine int_set
  subroutine rad_set (q)
    integer, intent(in) :: q
    type(flavor_t), dimension(3) :: flv
    type(quantum_numbers_t), dimension(3) :: qn
    call flavor_init (flv, (/ q, q, 21 /))
    call quantum_numbers_init (qn, flv)
    call interaction_add_state (rad, qn)
  end subroutine rad_set
end subroutine interaction_1
```

7.7 Matrix element evaluation

The `evaluator_t` type is an extension of the `interaction_t` type. It represents either a density matrix as the square of a transition matrix element, or the product of two density matrices. Usually, some quantum numbers are summed over in the result.

The `interaction_t` subobject represents a multi-particle interaction with incoming, virtual, and outgoing particles and the associated (not necessarily diagonal) density matrix of quantum state. When the evaluator is initialized, this interaction is constructed from the input interaction(s).

In addition, the initialization process sets up a multiplication table. For each matrix element of the result, it states which matrix elements are to be taken from the input interaction(s), multiplied (optionally, with an additional weight factor) and summed over.

Eventually, to a processes we associate a chain of evaluators which are to be evaluated sequentially. The physical event and its matrix element value(s) can be extracted from the last evaluator in such a chain.

Evaluators are constructed only once (as long as this is possible) during an initialization step. Then, for each event, momenta are computed and transferred among evaluators using the links within the interaction subobject. The multiplication tables enable fast evaluation of the result without looking at quantum numbers anymore.

⟨`evaluators.f90`⟩≡
  ⟨*File header*⟩

  ```
module evaluators
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use limits, only: FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices
    use interactions
```

  ⟨*Standard module head*⟩

  ⟨*Evaluators: public*⟩

  ⟨*Evaluators: parameters*⟩

  ⟨*Evaluators: types*⟩

  ⟨*Evaluators: interfaces*⟩

```
    contains

⟨Evaluators: procedures⟩

⟨Evaluators: tests⟩

    end module evaluators
```

7.7.1 Array of pairings

The evaluator contains an array of `pairing_array` objects. This makes up the multiplication table.

Each pairing array contains two list of matrix element indices and a list of numerical factors. The matrix element indices correspond to the input interactions. The corresponding matrix elements are to be multiplied and optionally multiplied by a factor. The results are summed over to yield one specific matrix element of the result evaluator.

⟨Evaluators: types⟩≡
```
    type :: pairing_array_t
       integer, dimension(:), allocatable :: i1, i2
       complex(default), dimension(:), allocatable :: factor
    end type pairing_array_t
```

⟨Evaluators: procedures⟩≡
```
    elemental subroutine pairing_array_init (pa, n, has_i2, has_factor)
      type(pairing_array_t), intent(out) :: pa
      integer, intent(in) :: n
      logical, intent(in) :: has_i2, has_factor
      allocate (pa%i1 (n))
      if (has_i2)  allocate (pa%i2 (n))
      if (has_factor)  allocate (pa%factor (n))
    end subroutine pairing_array_init
```

7.7.2 The evaluator type

Possible variants of evaluators:

⟨Evaluators: parameters⟩≡
```
    integer, parameter :: &
         EVAL_UNDEFINED = 0, &
         EVAL_PRODUCT = 1, &
         EVAL_SQUARED_FLOWS = 2, &
         EVAL_SQUARE_WITH_COLOR_FACTORS = 3, &
         EVAL_COLOR_CONTRACTION = 4, &
         EVAL_IDENTITY = 5, &
         EVAL_QN_SUM = 6
```

The evaluator type contains the result interaction and an array of pairing lists, one for each matrix element in the result interaction.

⟨Evaluators: public⟩≡
```
    public :: evaluator_t
```

⟨*Evaluators: types*⟩+≡
```
type :: evaluator_t
   private
   integer :: type = EVAL_UNDEFINED
   type(interaction_t), pointer :: int_in1 => null ()
   type(interaction_t), pointer :: int_in2 => null ()
   type(interaction_t) :: int
   type(pairing_array_t), dimension(:), allocatable :: pairing_array
 contains
 ⟨Evaluators: evaluator: TBP⟩
end type evaluator_t
```

Output.

⟨*Evaluators: evaluator: TBP*⟩≡
```
procedure :: write => evaluator_write
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine evaluator_write (eval, unit, &
     verbose, show_momentum_sum, show_mass, show_state, show_table, testflag)
  class(evaluator_t), intent(in) :: eval
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
  logical, intent(in), optional :: show_state, show_table, testflag
  logical :: conjugate, square, show_tab
  integer :: u, i, j
  u = output_unit (unit);  if (u < 0)  return
  show_tab = .true.;  if (present (show_table))  show_tab = .false.
  ! write (u, "(1x,A)")  "Evaluator:"      !!! Debugging
  call interaction_write &
       (eval%int, unit, verbose, show_momentum_sum, show_mass, &
          show_state, testflag)
  if (show_tab) then
     write (u, "(1x,A)")  "Matrix-element multiplication"
     write (u, "(2x,A)", advance="no")  "Input interaction 1:"
     if (associated (eval%int_in1)) then
        write (u, "(1x,I0)")  interaction_get_tag (eval%int_in1)
     else
        write (u, "(A)")  " [undefined]"
     end if
     write (u, "(2x,A)", advance="no")  "Input interaction 2:"
     if (associated (eval%int_in2)) then
        write (u, "(1x,I0)")  interaction_get_tag (eval%int_in2)
     else
        write (u, "(A)")  " [undefined]"
     end if
     select case (eval%type)
     case (EVAL_SQUARED_FLOWS, EVAL_SQUARE_WITH_COLOR_FACTORS)
        conjugate = .true.
        square = .true.
     case default
        conjugate = .false.
        square = .false.
     end select
     if (eval%type == EVAL_IDENTITY) then
```

```
            write (u, "(1X,A)") "Identity evaluator, pairing array unused"
            return
         end if
      if (allocated (eval%pairing_array)) then
         do i = 1, size (eval%pairing_array)
            write (u, "(2x,A,I0,A)")  "ME(", i, ") = "
            do j = 1, size (eval%pairing_array(i)%i1)
               write (u, "(4x,A)", advance="no")  "+"
               if (allocated (eval%pairing_array(i)%i2)) then
                  write (u, "(1x,A,I0,A)", advance="no")  &
                      "ME1(", eval%pairing_array(i)%i1(j), ")"
                  if (conjugate) then
                     write (u, "(A)", advance="no")  "* x"
                  else
                     write (u, "(A)", advance="no")  " x"
                  end if
                  write (u, "(1x,A,I0,A)", advance="no")  &
                      "ME2(", eval%pairing_array(i)%i2(j), ")"
               else if (square) then
                  write (u, "(1x,A)", advance="no")  "|"
                  write (u, "(A,I0,A)", advance="no")  &
                      "ME1(", eval%pairing_array(i)%i1(j), ")"
                  write (u, "(A)", advance="no")  "|^2"
               else
                  write (u, "(1x,A,I0,A)", advance="no")  &
                      "ME1(", eval%pairing_array(i)%i1(j), ")"
               end if
               if (allocated (eval%pairing_array(i)%factor)) then
                  write (u, "(1x,A)", advance="no")  "x"
                  write (u, "(1x,'(',"  // FMT_19 // ",',',"  // FMT_19 // &
                      ",')')") eval%pairing_array(i)%factor(j)
               else
                  write (u, *)
               end if
            end do
         end do
      end if
      ! print *, size (eval%pairing_array)        !!! Debugging
      end if
   end subroutine evaluator_write
```

Assignment: Deep copy of the interaction component.

⟨*Evaluators: public*⟩+≡

```
   public :: assignment(=)
```

⟨*Evaluators: interfaces*⟩≡

```
   interface assignment(=)
      module procedure evaluator_assign
   end interface
```

⟨*Evaluators: procedures*⟩+≡

```
   subroutine evaluator_assign (eval_out, eval_in)
     type(evaluator_t), intent(out) :: eval_out
     type(evaluator_t), intent(in) :: eval_in
```

```
      eval_out%type = eval_in%type
      eval_out%int_in1 => eval_in%int_in1
      eval_out%int_in2 => eval_in%int_in2
      eval_out%int = eval_in%int
      if (allocated (eval_in%pairing_array)) then
         allocate (eval_out%pairing_array (size (eval_in%pairing_array)))
         eval_out%pairing_array = eval_in%pairing_array
      end if
   end subroutine evaluator_assign
```

7.7.3 Auxiliary structures for evaluator creation

Creating an evaluator that properly handles all quantum numbers requires some bookkeeping. In this section, we define several auxiliary types and methods that organize and simplify this task. More structures are defined within the specific initializers (as local types and internal subroutines).

These types are currently implemented in a partial object-oriented way: We define some basic methods for initialization etc. here, but the evaluator routines below do access their internals as well. This simplifies some things such as index addressing using array slices, at the expense of losing some clarity.

Index mapping

Index mapping are abundant when constructing an evaluator. To have arrays of index mappings, we define this:

⟨*Evaluators: types*⟩+≡
```
   type :: index_map_t
      integer, dimension(:), allocatable :: entry
   end type index_map_t
```

⟨*Evaluators: procedures*⟩+≡
```
   elemental subroutine index_map_init (map, n)
     type(index_map_t), intent(out) :: map
     integer, intent(in) :: n
     allocate (map%entry (n))
   end subroutine index_map_init
```

⟨*Evaluators: procedures*⟩+≡
```
   function index_map_exists (map) result (flag)
     logical :: flag
     type(index_map_t), intent(in) :: map
     flag = allocated (map%entry)
   end function index_map_exists
```

⟨*Evaluators: interfaces*⟩+≡
```
   interface size
      module procedure index_map_size
   end interface
```

⟨*Evaluators: procedures*⟩+≡
```
function index_map_size (map) result (s)
  integer :: s
  type(index_map_t), intent(in) :: map
  if (allocated (map%entry)) then
     s = size (map%entry)
  else
     s = 0
  end if
end function index_map_size
```

⟨*Evaluators: interfaces*⟩+≡
```
interface assignment(=)
   module procedure index_map_assign_int
   module procedure index_map_assign_array
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
elemental subroutine index_map_assign_int (map, ival)
  type(index_map_t), intent(inout) :: map
  integer, intent(in) :: ival
  map%entry = ival
end subroutine index_map_assign_int

subroutine index_map_assign_array (map, array)
  type(index_map_t), intent(inout) :: map
  integer, dimension(:), intent(in) :: array
  map%entry = array
end subroutine index_map_assign_array
```

⟨*Evaluators: procedures*⟩+≡
```
elemental subroutine index_map_set_entry (map, i, ival)
  type(index_map_t), intent(inout) :: map
  integer, intent(in) :: i
  integer, intent(in) :: ival
  map%entry(i) = ival
end subroutine index_map_set_entry
```

⟨*Evaluators: procedures*⟩+≡
```
elemental function index_map_get_entry (map, i) result (ival)
  integer :: ival
  type(index_map_t), intent(in) :: map
  integer, intent(in) :: i
  ival = map%entry(i)
end function index_map_get_entry
```

Index mapping (two-dimensional)

This is a variant with a square matrix instead of an array.

⟨*Evaluators: types*⟩+≡
```
type :: index_map2_t
```

```
      integer :: s = 0
      integer, dimension(:,:), allocatable :: entry
    end type index_map2_t
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine index_map2_init (map, n)
      type(index_map2_t), intent(out) :: map
      integer, intent(in) :: n
      map%s = n
      allocate (map%entry (n, n))
    end subroutine index_map2_init
```

⟨*Evaluators: procedures*⟩+≡
```
    function index_map2_exists (map) result (flag)
      logical :: flag
      type(index_map2_t), intent(in) :: map
      flag = allocated (map%entry)
    end function index_map2_exists
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface size
       module procedure index_map2_size
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    function index_map2_size (map) result (s)
      integer :: s
      type(index_map2_t), intent(in) :: map
      s = map%s
    end function index_map2_size
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure index_map2_assign_int
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine index_map2_assign_int (map, ival)
      type(index_map2_t), intent(inout) :: map
      integer, intent(in) :: ival
      map%entry = ival
    end subroutine index_map2_assign_int
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine index_map2_set_entry (map, i, j, ival)
      type(index_map2_t), intent(inout) :: map
      integer, intent(in) :: i, j
      integer, intent(in) :: ival
      map%entry(i,j) = ival
    end subroutine index_map2_set_entry
```

⟨*Evaluators: procedures*⟩+≡

```
    elemental function index_map2_get_entry (map, i, j) result (ival)
      integer :: ival
      type(index_map2_t), intent(in) :: map
      integer, intent(in) :: i, j
      ival = map%entry(i,j)
    end function index_map2_get_entry
```

Auxiliary structures: particle mask

This is a simple container of a logical array.

⟨*Evaluators: types*⟩+≡

```
    type :: prt_mask_t
      logical, dimension(:), allocatable :: entry
    end type prt_mask_t
```

⟨*Evaluators: procedures*⟩+≡

```
    subroutine prt_mask_init (mask, n)
      type(prt_mask_t), intent(out) :: mask
      integer, intent(in) :: n
      allocate (mask%entry (n))
    end subroutine prt_mask_init
```

⟨*Evaluators: interfaces*⟩+≡

```
    interface size
      module procedure prt_mask_size
    end interface
```

⟨*Evaluators: procedures*⟩+≡

```
    function prt_mask_size (mask) result (s)
      integer :: s
      type(prt_mask_t), intent(in) :: mask
      s = size (mask%entry)
    end function prt_mask_size
```

Quantum number containers

Trivial transparent containers:

⟨*Evaluators: types*⟩+≡

```
    type :: qn_list_t
      type(quantum_numbers_t), dimension(:,:), allocatable :: qn
    end type qn_list_t

    type :: qn_mask_array_t
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
    end type qn_mask_array_t
```

Auxiliary structures: connection entries

This type is used as intermediate storage when computing the product of two evaluators or the square of an evaluator. The quantum-number array `qn` corresponds to the particles common to both interactions, but irrelevant quantum numbers (color) masked out. The index arrays `index_in` determine the entries in the input interactions that contribute to this connection. `n_index` is the size of these arrays, and `count` is used while filling the entries. Finally, the quantum-number arrays `qn_in_list` are the actual entries in the input interaction that contribute. In the product case, they exclude the connected quantum numbers.

Each evaluator has its own `connection_table` which contains an array of `connection_entry` objects, but also has stuff that specifically applies to the evaluator type. Hence, we do not generalize the `connection_table_t` type.

The filling procedure `connection_entry_add_state` is specific to the various evaluator types.

⟨*Evaluators: types*⟩+≡
```
type :: connection_entry_t
   type(quantum_numbers_t), dimension(:), allocatable :: qn_conn
   integer, dimension(:), allocatable :: n_index
   integer, dimension(:), allocatable :: count
   type(index_map_t), dimension(:), allocatable :: index_in
   type(qn_list_t), dimension(:), allocatable :: qn_in_list
end type connection_entry_t
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine connection_entry_init &
    (entry, n_count, n_map, qn_conn, count, n_rest)
  type(connection_entry_t), intent(out) :: entry
  integer, intent(in) :: n_count, n_map
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_conn
  integer, dimension(n_count), intent(in) :: count
  integer, dimension(n_count), intent(in) :: n_rest
  integer :: i
  allocate (entry%qn_conn (size (qn_conn)))
  allocate (entry%n_index (n_count))
  allocate (entry%count (n_count))
  allocate (entry%index_in (n_map))
  allocate (entry%qn_in_list (n_count))
  entry%qn_conn = qn_conn
  entry%n_index = count
  entry%count = 0
  if (size (entry%index_in) == size (count)) then
     call index_map_init (entry%index_in, count)
  else
     call index_map_init (entry%index_in, count(1))
  end if
  do i = 1, n_count
     allocate (entry%qn_in_list(i)%qn (n_rest(i), count(i)))
  end do
end subroutine connection_entry_init
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine connection_entry_write (entry, unit)
  type(connection_entry_t), intent(in) :: entry
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = output_unit (unit)
  call quantum_numbers_write (entry%qn_conn, unit)
  write (u, *)
  do i = 1, size (entry%n_index)
     write (u, *)  "Input interaction", i
     do j = 1, entry%n_index(i)
        if (size (entry%n_index) == size (entry%index_in)) then
           write (u, "(2x,I0,4x,I0,2x)", advance = "no") &
                j, index_map_get_entry (entry%index_in(i), j)
        else
           write (u, "(2x,I0,4x,I0,2x,I0,2x)", advance = "no") &
                j, index_map_get_entry (entry%index_in(1), j), &
                   index_map_get_entry (entry%index_in(2), j)
        end if
        call quantum_numbers_write (entry%qn_in_list(i)%qn(:,j), unit)
        write (u, *)
     end do
  end do
end subroutine connection_entry_write
```

Color handling

For managing color-factor computation, we introduce this local type. The `index`
is the index in the color table that corresponds to a given matrix element index
in the input interaction. The `col` array stores the color assignments in rows.
The `factor` array associates a complex number with each pair of arrays in the
color table. The `factor_is_known` array reveals whether a given factor is known
already or still has to be computed.

⟨*Evaluators: types*⟩+≡
```
type :: color_table_t
   integer, dimension(:), allocatable :: index
   type(color_t), dimension(:,:), allocatable :: col
   logical, dimension(:,:), allocatable :: factor_is_known
   complex(default), dimension(:,:), allocatable :: factor
end type color_table_t
```

This is the initializer. We extract the color states from the given state matrices,
establish index mappings between the two states (implemented by the array
`me_index`), make an array of color states, and initialize the color-factor table.
The latter is two-dimensional (includes interference) and not yet filled.

⟨*Evaluators: procedures*⟩+≡
```
subroutine color_table_init (color_table, state, n_tot)
  type(color_table_t), intent(out) :: color_table
  type(state_matrix_t), intent(in) :: state
  integer, intent(in) :: n_tot
  type(state_iterator_t) :: it
```

```
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      type(state_matrix_t) :: state_col
      integer :: index, n_col_state
      allocate (color_table%index &
          (state_matrix_get_n_matrix_elements (state)))
      color_table%index = 0
      allocate (qn (n_tot))
      call state_matrix_init (state_col)
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         index = state_iterator_get_me_index (it)
         call quantum_numbers_init (qn, col = state_iterator_get_color (it))
         call state_matrix_add_state (state_col, qn, &
             me_index = color_table%index(index))
         call state_iterator_advance (it)
      end do
      n_col_state = state_matrix_get_n_matrix_elements (state_col)
      allocate (color_table%col (n_tot, n_col_state))
      call state_iterator_init (it, state_col)
      do while (state_iterator_is_valid (it))
         index = state_iterator_get_me_index (it)
         color_table%col(:,index) = state_iterator_get_color (it)
         call state_iterator_advance (it)
      end do
      call state_matrix_final (state_col)
      allocate (color_table%factor_is_known (n_col_state, n_col_state))
      allocate (color_table%factor (n_col_state, n_col_state))
      color_table%factor_is_known = .false.
    end subroutine color_table_init
```

Output (debugging use):

⟨*Evaluators: procedures*⟩+≡

```
    subroutine color_table_write (color_table, unit)
      type(color_table_t), intent(in) :: color_table
      integer, intent(in), optional :: unit
      integer :: i, j
      integer :: u
      u = output_unit (unit)
      write (u, *) "Color table:"
      if (allocated (color_table%index)) then
         write (u, *) "  Index mapping state => color table:"
         do i = 1, size (color_table%index)
            write (u, "(3x,I0,2x,I0,2x)")  i, color_table%index(i)
         end do
         write (u, *) "  Color table:"
         do i = 1, size (color_table%col, 2)
            write (u, "(3x,I0,2x)", advance = "no")  i
            call color_write (color_table%col(:,i), unit)
            write (u, *)
         end do
         write (u, *) "  Defined color factors:"
         do i = 1, size (color_table%factor, 1)
            do j = 1, size (color_table%factor, 2)
                if (color_table%factor_is_known(i,j)) then
```

```
                write (u, *)  i, j, color_table%factor(i,j)
              end if
          end do
        end do
      end if
  end subroutine color_table_write
```

This subroutine sets color factors, based on information from the hard matrix element: the list of pairs of color-flow indices (in the basis of the matrix element code), the list of corresponding factors, and the list of mappings from the matrix element index in the input interaction to the color-flow index in the hard matrix element object.

We first determine the mapping of color-flow indices from the hard matrix element code to the current color table. The mapping could be nontrivial because the latter is derived from iterating over a state matrix, which may return states in non-canonical order. The translation table can be determined because we have, for the complete state matrix, both the mapping to the hard interaction (the input `col_index_hi`) and the mapping to the current color table (the component `color_table%index`).

Once this mapping is known, we scan the list of index pairs `color_flow_index` and translate them to valid color-table index pairs. For this pair, the color factor is set using the corresponding entry in the list `col_factor`.

⟨*Evaluators: procedures*⟩+≡
```
  subroutine color_table_set_color_factors (color_table, &
      col_flow_index, col_factor, col_index_hi)
    type(color_table_t), intent(inout) :: color_table
    integer, dimension(:,:), intent(in) :: col_flow_index
    complex(default), dimension(:), intent(in) :: col_factor
    integer, dimension(:), intent(in) :: col_index_hi
    integer, dimension(:), allocatable :: hi_to_ct
    integer :: n_cflow
    integer :: hi_index, me_index, ct_index, cf_index
    integer, dimension(2) :: hi_index_pair, ct_index_pair
    n_cflow = size (col_index_hi)
    if (size (color_table%index) /= n_cflow) &
        call msg_bug ("Mismatch between hard matrix element and color table")
    allocate (hi_to_ct (n_cflow))
    do me_index = 1, size (color_table%index)
       ct_index = color_table%index(me_index)
       hi_index = col_index_hi(me_index)
       hi_to_ct(hi_index) = ct_index
    end do
    do cf_index = 1, size (col_flow_index, 2)
       hi_index_pair = col_flow_index(:,cf_index)
       ct_index_pair = hi_to_ct(hi_index_pair)
       color_table%factor(ct_index_pair(1), ct_index_pair(2)) = &
            col_factor(cf_index)
       color_table%factor_is_known(ct_index_pair(1), ct_index_pair(2)) = .true.
    end do
  end subroutine color_table_set_color_factors
```

This function returns a color factor, given two indices which point to the matrix

elements of the initial state matrix. Internally, we can map them to the corresponding indices in the color table. As a side effect, we store the color factor in the color table for later lookup. (I.e., this function is impure.)

⟨*Evaluators: procedures*⟩+≡

```
function color_table_get_color_factor (color_table, index1, index2, nc) &
     result (factor)
  real(default) :: factor
  type(color_table_t), intent(inout) :: color_table
  integer, intent(in) :: index1, index2
  integer, intent(in), optional :: nc
  integer :: i1, i2
  ! print *, "compute color factor ", index1, index2   !!! Debugging
  i1 = color_table%index(index1)
  i2 = color_table%index(index2)
  ! print *, "  indices = ", i1, i2                      !!! Debugging
  if (color_table%factor_is_known(i1,i2)) then
     factor = color_table%factor(i1,i2)
     ! print *, "  is known : ", factor                  !!! Debugging
  else
     factor = compute_color_factor &
          (color_table%col(:,i1), color_table%col(:,i2), nc)
     color_table%factor(i1,i2) = factor
     color_table%factor_is_known(i1,i2) = .true.
     ! print *, "  computed : ", factor                  !!! Debugging
  end if
 end function color_table_get_color_factor
```

7.7.4 Creating an evaluator: Matrix multiplication

The evaluator for matrix multiplication is the most complicated variant.

The initializer takes two input interactions and constructs the result evaluator, which consists of the interaction and the multiplication table for the product (or convolution) of the two. Normally, the input interactions are connected by one or more common particles (e.g., decay, structure function convolution).

In the result interaction, quantum numbers of the connections can be summed over. This is determined by the `qn_mask_conn` argument. The `qn_mask_rest` argument is its analog for the other particles within the result interaction. (E.g., for the trace of the state matrix, all quantum numbers are summed over.) Finally, the `connections_are_resonant` argument tells whether the connecting particles should be marked as resonant in the final event record. This is useful for decays.

The algorithm consists of the following steps:

1. `find_connections`: Find the particles which are connected, i.e., common to both input interactions. Either they are directly linked, or both are linked to a common source.

2. `compute_index_bounds_and_mappings`: Compute the mappings of particle indices from the input interactions to the result interaction. There is a separate mapping for the connected particles.

834

3. `accumulate_connected_states`: Create an auxiliary state matrix which lists the possible quantum numbers for the connected particles. When building this matrix, count the number of times each assignment is contained in any of the input states and, for each of the input states, record the index of the matrix element within the new state matrix. For the connected particles, reassign color indices such that no color state is present twice in different color-index assignment. Note that helicity assignments of the connected state can be (and will be) off-diagonal, so no spin correlations are lost in decays.

   Do this for both input interactions.

4. `allocate_connection_entries`: Allocate a table of connections. Each table row corresponds to one state in the auxiliary matrix, and to multiple states of the input interactions. It collects all states of the unconnected particles in the two input interactions that are associated with the particular state (quantum-number assignment) of the connected particles.

5. `fill_connection_table`: Fill the table of connections by scanning both input interactions. When copying states, reassign color indices for the unconnected particles such that they match between all involved particle sets (interaction 1, interaction 2, and connected particles).

6. `make_product_interaction`: Scan the table of connections we have just built. For each entry, construct all possible pairs of states of the unconnected particles and combine them with the specific connected-particle state. This is a possible quantum-number assignment of the result interaction. Now mask all quantum numbers that should be summed over, and append this to the result state matrix. Record the matrix element index of the result. We now have the result interaction.

7. `make_pairing_array`: First allocate the pairing array with the number of entries of the result interaction. Then scan the table of connections again. For each entry, record the indices of the matrix elements which have to be multiplied and summed over in order to compute this particular matrix element. This makes up the multiplication table.

8. `record_links`: Transfer all source pointers from the input interactions to the result interaction. Do the same for the internal parent-child relations and resonance assignments. For the connected particles, make up appropriate additional parent-child relations. This allows for fetching momenta from other interactions when a new event is filled, and to reconstruct the event history when the event is analyzed.

After all this is done, for each event, we just have to evaluate the pairing arrays (multiplication tables) in order to compute the result matrix elements in their proper positions. The quantum-number assignments remain fixed from now on.

⟨*Evaluators: public*⟩+≡
```
public :: evaluator_init_product
```

⟨*Evaluators: interfaces*⟩+≡
```
interface evaluator_init_product
   module procedure evaluator_init_product_ii
   module procedure evaluator_init_product_ie
   module procedure evaluator_init_product_ei
   module procedure evaluator_init_product_ee
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine evaluator_init_product_ii &
     (eval, int_in1, int_in2, qn_mask_conn, qn_mask_rest, &
      connections_are_resonant)

  type(evaluator_t), intent(out), target :: eval
  type(interaction_t), intent(in), target :: int_in1, int_in2
  type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
  type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
  logical, intent(in), optional :: connections_are_resonant

  type(qn_mask_array_t), dimension(2) :: qn_mask_in

  type :: connection_table_t
     integer :: n_conn = 0
     integer, dimension(2) :: n_rest = 0
     integer :: n_tot = 0
     integer :: n_me_conn = 0
     type(state_matrix_t) :: state
     type(index_map_t), dimension(:), allocatable :: index_conn
     type(connection_entry_t), dimension(:), allocatable :: entry
     type(index_map_t) :: index_result
  end type connection_table_t
  type(connection_table_t) :: connection_table

  integer :: n_in, n_vir, n_out, n_tot
  integer, dimension(2) :: n_rest
  integer :: n_conn

  integer, dimension(:,:), allocatable :: connection_index
  type(index_map_t), dimension(2) :: prt_map_in
  type(index_map_t) :: prt_map_conn
  type(prt_mask_t), dimension(2) :: prt_is_connected
  type(quantum_numbers_mask_t), dimension(:), allocatable :: &
       qn_mask_conn_initial

  integer :: i

  eval%type = EVAL_PRODUCT
  eval%int_in1 => int_in1
  eval%int_in2 => int_in2
  ! print *, "Evaluator product"         !!! Debugging
  ! print *, "First interaction"         !!! Debugging
  ! call interaction_write (int_in1)     !!! Debugging
  ! print *                              !!! Debugging
  ! print *, "Second interaction"        !!! Debugging
```

```
! call interaction_write (int_in2)         !!! Debugging
! print *                                  !!! Debugging

call find_connections (int_in1, int_in2, n_conn, connection_index)
if (n_conn == 0) then
   call msg_message ("First interaction:")
   call interaction_write (int_in1)
   call msg_message ("Second interaction:")
   call interaction_write (int_in2)
   call msg_fatal ("Evaluator product: no connections found between factors")
end if
call compute_index_bounds_and_mappings &
     (int_in1, int_in2, n_conn, &
      n_in, n_vir, n_out, n_tot, &
      n_rest, prt_map_in, prt_map_conn)

call prt_mask_init (prt_is_connected(1), interaction_get_n_tot (int_in1))
call prt_mask_init (prt_is_connected(2), interaction_get_n_tot (int_in2))
do i = 1, 2
   prt_is_connected(i)%entry = .true.
   prt_is_connected(i)%entry(connection_index(:,i)) = .false.
end do
allocate (qn_mask_conn_initial (n_conn))
qn_mask_conn_initial = &
     interaction_get_mask (int_in1, connection_index(:,1)) .or. &
     interaction_get_mask (int_in2, connection_index(:,2))
allocate (qn_mask_in(1)%mask (interaction_get_n_tot (int_in1)))
allocate (qn_mask_in(2)%mask (interaction_get_n_tot (int_in2)))
qn_mask_in(1)%mask = interaction_get_mask (int_in1)
qn_mask_in(2)%mask = interaction_get_mask (int_in2)

call connection_table_init (connection_table, &
     interaction_get_state_matrix_ptr (int_in1), &
     interaction_get_state_matrix_ptr (int_in2), &
     qn_mask_conn_initial,  &
     n_conn, connection_index, n_rest)
call connection_table_fill (connection_table, &
     interaction_get_state_matrix_ptr (int_in1), &
     interaction_get_state_matrix_ptr (int_in2), &
     connection_index, prt_is_connected)
call make_product_interaction (eval%int, &
     n_in, n_vir, n_out, &
     connection_table, &
     prt_map_in, prt_is_connected, &
     qn_mask_in, qn_mask_conn_initial, qn_mask_conn, qn_mask_rest)
! call connection_table_write (connection_table)     !!! Debugging
call make_pairing_array (eval%pairing_array, &
     interaction_get_n_matrix_elements (eval%int), &
     connection_table)
call record_links (eval%int, &
     int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
     prt_is_connected, connections_are_resonant)
call connection_table_final (connection_table)
```

```
      ! print *, "Result evaluator"                   !!! Debugging
      ! call evaluator_write (eval)                    !!! Debugging

    if (interaction_get_n_matrix_elements (eval%int) == 0) then
       print *, "Evaluator product"
       print *, "First interaction"
       call interaction_write (int_in1)
       print *
       print *, "Second interaction"
       call interaction_write (int_in2)
       print *
       call msg_fatal ("Product of density matrices is empty", &
            [var_str ("   ------------------------------------------"), &
             var_str ("This happens when two density matrices are convoluted "), &
             var_str ("but the processes they belong to (e.g., production "), &
             var_str ("and decay) do not match. This could happen if the "), &
             var_str ("beam specification does not match the hard "), &
             var_str ("process. Or it may indicate a WHIZARD bug.")])
    end if

contains

  subroutine compute_index_bounds_and_mappings &
        (int1, int2, n_conn, &
         n_in, n_vir, n_out, n_tot, &
         n_rest, prt_map_in, prt_map_conn)
    type(interaction_t), intent(in) :: int1, int2
    integer, intent(in) :: n_conn
    integer, intent(out) :: n_in, n_vir, n_out, n_tot
    integer, dimension(2), intent(out) :: n_rest
    type(index_map_t), dimension(2), intent(out) :: prt_map_in
    type(index_map_t), intent(out) :: prt_map_conn
    integer, dimension(:), allocatable :: index
    integer :: n_in1, n_vir1, n_out1
    integer :: n_in2, n_vir2, n_out2
    integer :: k
    n_in1  = interaction_get_n_in  (int1)
    n_vir1 = interaction_get_n_vir (int1)
    n_out1 = interaction_get_n_out (int1) - n_conn
    n_rest(1) = n_in1 + n_vir1 + n_out1
    n_in2  = interaction_get_n_in  (int2) - n_conn
    n_vir2 = interaction_get_n_vir (int2)
    n_out2 = interaction_get_n_out (int2)
    n_rest(2) = n_in2 + n_vir2 + n_out2
    n_in  = n_in1  + n_in2
    n_vir = n_vir1 + n_vir2 + n_conn
    n_out = n_out1 + n_out2
    n_tot = n_in + n_vir + n_out
    call index_map_init (prt_map_in, n_rest)
    call index_map_init (prt_map_conn, n_conn)
    allocate (index (n_tot))
    index = (/ (i, i = 1, n_tot) /)
    prt_map_in(1)%entry(1 : n_in1) = index(  1 :   n_in1)
    k =      n_in1
```

```fortran
    prt_map_in(2)%entry(1 : n_in2) = index(k+1 : k+n_in2)
    k = k + n_in2
    prt_map_in(1)%entry(n_in1+1 : n_in1+n_vir1) = index(k+1 : k+n_vir1)
    k = k + n_vir1
    prt_map_in(2)%entry(n_in2+1 : n_in2+n_vir2) = index(k+1 : k+n_vir2)
    k = k + n_vir2
    prt_map_conn%entry = index(k+1 : k+n_conn)
    k = k + n_conn
    prt_map_in(1)%entry(n_in1+n_vir1+1 : n_rest(1)) = index(k+1 : k+n_out1)
    k = k + n_out1
    prt_map_in(2)%entry(n_in2+n_vir2+1 : n_rest(2)) = index(k+1 : k+n_out2)
end subroutine compute_index_bounds_and_mappings

subroutine connection_table_init &
      (connection_table, state_in1, state_in2, qn_mask_conn, &
       n_conn, connection_index, n_rest)
   type(connection_table_t), intent(out) :: connection_table
   type(state_matrix_t), intent(in), target :: state_in1, state_in2
   type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_conn
   integer, intent(in) :: n_conn
   integer, dimension(:,:), intent(in) :: connection_index
   integer, dimension(2), intent(in) :: n_rest
   integer, dimension(2) :: n_me_in
   type(state_iterator_t) :: it
   type(quantum_numbers_t), dimension(n_conn) :: qn
   integer :: i, me_index_in, me_index_conn, n_me_conn
   integer, dimension(2) :: me_count
   connection_table%n_conn = n_conn
   connection_table%n_rest = n_rest
   n_me_in(1) = state_matrix_get_n_matrix_elements (state_in1)
   n_me_in(2) = state_matrix_get_n_matrix_elements (state_in2)
   allocate (connection_table%index_conn (2))
   call index_map_init (connection_table%index_conn, n_me_in)
   connection_table%index_conn = 0
   call state_matrix_init (connection_table%state, n_counters=2)
   do i = 1, 2
      select case (i)
      case (1);  call state_iterator_init (it, state_in1)
      case (2);  call state_iterator_init (it, state_in2)
      end select
      do while (state_iterator_is_valid (it))
         qn = state_iterator_get_quantum_numbers (it, connection_index(:,i))
         call quantum_numbers_undefine (qn, qn_mask_conn)
         call quantum_numbers_canonicalize_color (qn)
         me_index_in = state_iterator_get_me_index (it)
         call state_matrix_add_state (connection_table%state, qn, &
              counter_index = i, me_index = me_index_conn)
         call index_map_set_entry (connection_table%index_conn(i), &
              me_index_in, me_index_conn)
         call state_iterator_advance (it)
      end do
   end do
   n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
   connection_table%n_me_conn = n_me_conn
```

```fortran
    allocate (connection_table%entry (n_me_conn))
    call state_iterator_init (it, connection_table%state)
    do while (state_iterator_is_valid (it))
       i = state_iterator_get_me_index (it)
       me_count = state_iterator_get_me_count (it)
       call connection_entry_init (connection_table%entry(i), 2, 2, &
            state_iterator_get_quantum_numbers (it), &
            me_count, n_rest)
       call state_iterator_advance (it)
    end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
  type(connection_table_t), intent(inout) :: connection_table
  call state_matrix_final (connection_table%state)
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
  type(connection_table_t), intent(in) :: connection_table
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = output_unit (unit)
  write (u, *) "Connection table:"
  call state_matrix_write (connection_table%state, unit)
  if (allocated (connection_table%index_conn)) then
     write (u, *) "  Index mapping input => connection table:"
     do i = 1, size (connection_table%index_conn)
        write (u, *) "  Input state", i
        do j = 1, size (connection_table%index_conn(i))
           write (u, *)  j, &
                index_map_get_entry (connection_table%index_conn(i), j)
        end do
     end do
  end if
  if (allocated (connection_table%entry)) then
     write (u, *) "  Connection table contents:"
     do i = 1, size (connection_table%entry)
        call connection_entry_write (connection_table%entry(i), unit)
     end do
  end if
  if (index_map_exists (connection_table%index_result)) then
     write (u, *) "  Index mapping connection table => output:"
     do i = 1, size (connection_table%index_result)
        write (u, *)  i, &
             index_map_get_entry (connection_table%index_result, i)
     end do
  end if
end subroutine connection_table_write

subroutine connection_table_fill &
    (connection_table, state_in1, state_in2, &
     connection_index, prt_is_connected)
  type(connection_table_t), intent(inout) :: connection_table
```

```
    type(state_matrix_t), intent(in), target :: state_in1, state_in2
    integer, dimension(:,:), intent(in) :: connection_index
    type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
    type(state_iterator_t) :: it
    integer :: index_in, index_conn
    integer :: color_offset
    integer :: n_result_entries
    integer :: i, k
    color_offset = state_matrix_get_max_color_value (connection_table%state)
    do i = 1, 2
       select case (i)
       case (1);  call state_iterator_init (it, state_in1)
       case (2);  call state_iterator_init (it, state_in2)
       end select
       do while (state_iterator_is_valid (it))
          index_in = state_iterator_get_me_index (it)
          index_conn = index_map_get_entry &
                          (connection_table%index_conn(i), index_in)
          if (index_conn /= 0) then
             call connection_entry_add_state &
                  (connection_table%entry(index_conn), i, &
                   index_in, &
                   state_iterator_get_quantum_numbers (it), &
                   connection_index(:,i), prt_is_connected(i), &
                   color_offset)
          end if
          call state_iterator_advance (it)
       end do
       color_offset = color_offset &
            + state_matrix_get_max_color_value (state_in1)
    end do
    n_result_entries = 0
    do k = 1, size (connection_table%entry)
       n_result_entries = &
            n_result_entries + product (connection_table%entry(k)%n_index)
    end do
    call index_map_init (connection_table%index_result, n_result_entries)
  end subroutine connection_table_fill

  subroutine connection_entry_add_state &
      (entry, i, index_in, qn_in, connection_index, prt_is_connected, &
       color_offset)
    type(connection_entry_t), intent(inout) :: entry
    integer, intent(in) :: i
    integer, intent(in) :: index_in
    type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
    integer, dimension(:), intent(in) :: connection_index
    type(prt_mask_t), intent(in) :: prt_is_connected
    integer, intent(in) :: color_offset
    integer :: c
    integer, dimension(:,:), allocatable :: color_map
    entry%count(i) = entry%count(i) + 1
    c = entry%count(i)
    call quantum_numbers_set_color_map &
```

841

```
        (color_map, qn_in(connection_index), entry%qn_conn)
    call index_map_set_entry (entry%index_in(i), c, index_in)
    entry%qn_in_list(i)%qn(:,c) = pack (qn_in, prt_is_connected%entry)
    call quantum_numbers_translate_color &
        (entry%qn_in_list(i)%qn(:,c), color_map, color_offset)
end subroutine connection_entry_add_state

subroutine make_product_interaction (int, &
      n_in, n_vir, n_out, &
      connection_table, &
      prt_map_in, prt_is_connected, &
      qn_mask_in, qn_mask_conn_initial, qn_mask_conn, qn_mask_rest)
    type(interaction_t), intent(out), target :: int
    integer, intent(in) :: n_in, n_vir, n_out
    type(connection_table_t), intent(inout), target :: connection_table
    type(index_map_t), dimension(2), intent(in) :: prt_map_in
    type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
    type(qn_mask_array_t), dimension(2), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: &
        qn_mask_conn_initial
    type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
    type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
    type(index_map_t), dimension(2) :: prt_index_in
    type(index_map_t) :: prt_index_conn
    integer :: n_tot, n_conn
    integer, dimension(2) :: n_rest
    integer :: i, j, k, m
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    type(connection_entry_t), pointer :: entry
    integer :: result_index
    n_conn = connection_table%n_conn
    n_rest = connection_table%n_rest
    n_tot = sum (n_rest) + n_conn
    allocate (qn (n_tot), qn_mask (n_tot))
    do i = 1, 2
       call index_map_init (prt_index_in(i), n_rest(i))
       prt_index_in(i) = &
           prt_map_in(i)%entry ((/ (j, j = 1, n_rest(i)) /))
    end do
    call index_map_init (prt_index_conn, n_conn)
    prt_index_conn = prt_map_conn%entry ((/ (j, j = 1, n_conn) /))
    do i = 1, 2
       if (present (qn_mask_rest)) then
          qn_mask(prt_index_in(i)%entry) = &
              pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry) &
              .or. qn_mask_rest
       else
          qn_mask(prt_index_in(i)%entry) = &
              pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry)
       end if
    end do
    qn_mask(prt_index_conn%entry) = qn_mask_conn_initial .or. qn_mask_conn
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
```

```
      m = 1
      do i = 1, connection_table%n_me_conn
         entry => connection_table%entry(i)
         qn(prt_index_conn%entry) = &
              quantum_numbers_undefined (entry%qn_conn, qn_mask_conn)
         do j = 1, entry%n_index(1)
            qn(prt_index_in(1)%entry) = entry%qn_in_list(1)%qn(:,j)
            do k = 1, entry%n_index(2)
               qn(prt_index_in(2)%entry) = entry%qn_in_list(2)%qn(:,k)
               call interaction_add_state (int, qn, me_index = result_index)
               call index_map_set_entry &
                    (connection_table%index_result, m, result_index)
               m = m + 1
            end do
         end do
      end do
      call interaction_freeze (int)
   end subroutine make_product_interaction

   subroutine make_pairing_array (pa, n_matrix_elements, connection_table)
      type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
      integer, intent(in) :: n_matrix_elements
      type(connection_table_t), intent(in), target :: connection_table
      type(connection_entry_t), pointer :: entry
      integer, dimension(:), allocatable :: n_entries
      integer :: i, j, k, m, r
      allocate (pa (n_matrix_elements))
      allocate (n_entries (n_matrix_elements))
      n_entries = 0
      do m = 1, size (connection_table%index_result)
         r = index_map_get_entry (connection_table%index_result, m)
         n_entries(r) = n_entries(r) + 1
      end do
      call pairing_array_init &
           (pa, n_entries, has_i2=.true., has_factor=.false.)
      m = 1
      n_entries = 0
      do i = 1, connection_table%n_me_conn
         entry => connection_table%entry(i)
         do j = 1, entry%n_index(1)
            do k = 1, entry%n_index(2)
               r = index_map_get_entry (connection_table%index_result, m)
               n_entries(r) = n_entries(r) + 1
               pa(r)%i1(n_entries(r)) = &
                    index_map_get_entry (entry%index_in(1), j)
               pa(r)%i2(n_entries(r)) = &
                    index_map_get_entry (entry%index_in(2), k)
               m = m + 1
            end do
         end do
      end do
   end subroutine make_pairing_array

   subroutine record_links (int, &
```

```
        int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
        prt_is_connected, connections_are_resonant)
    type(interaction_t), intent(inout) :: int
    type(interaction_t), intent(in), target :: int_in1, int_in2
    integer, dimension(:,:), intent(in) :: connection_index
    type(index_map_t), dimension(2), intent(in) :: prt_map_in
    type(index_map_t), intent(in) :: prt_map_conn
    type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
    logical, intent(in), optional :: connections_are_resonant
    type(index_map_t), dimension(2) :: prt_map_all
    integer :: i, j, k, ival
    call index_map_init (prt_map_all(1), size (prt_is_connected(1)))
    k = 0
    j = 0
    do i = 1, size (prt_is_connected(1))
       if (prt_is_connected(1)%entry(i)) then
          j = j + 1
          ival = index_map_get_entry (prt_map_in(1), j)
          call index_map_set_entry (prt_map_all(1), i, ival)
       else
          k = k + 1
          ival = index_map_get_entry (prt_map_conn, k)
          call index_map_set_entry (prt_map_all(1), i, ival)
       end if
       call interaction_set_source_link (int, ival, int_in1, i)
    end do
    call interaction_transfer_relations (int_in1, int, prt_map_all(1)%entry)
    call index_map_init (prt_map_all(2), size (prt_is_connected(2)))
    j = 0
    do i = 1, size (prt_is_connected(2))
       if (prt_is_connected(2)%entry(i)) then
          j = j + 1
          ival = index_map_get_entry (prt_map_in(2), j)
          call index_map_set_entry (prt_map_all(2), i, ival)
          call interaction_set_source_link (int, ival, int_in2, i)
       else
          call index_map_set_entry (prt_map_all(2), i, 0)
       end if
    end do
    call interaction_transfer_relations (int_in2, int, prt_map_all(2)%entry)
    call interaction_relate_connections (int, &
         int_in2, connection_index(:,2), prt_map_all(2)%entry, &
         prt_map_conn%entry, connections_are_resonant)
  end subroutine record_links

end subroutine evaluator_init_product_ii

subroutine evaluator_init_product_ie &
     (eval, int_in1, eval_in2, qn_mask_conn, qn_mask_rest, &
      connections_are_resonant)
  type(evaluator_t), intent(out), target :: eval
  type(interaction_t), intent(in), target :: int_in1
  type(evaluator_t), intent(in), target :: eval_in2
  type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
```

```
      type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
      logical, intent(in), optional :: connections_are_resonant
      call evaluator_init_product_ii &
           (eval, int_in1, eval_in2%int, qn_mask_conn, qn_mask_rest, &
            connections_are_resonant)
    end subroutine evaluator_init_product_ie

    subroutine evaluator_init_product_ei &
         (eval, eval_in1, int_in2, qn_mask_conn, qn_mask_rest, &
          connections_are_resonant)
      type(evaluator_t), intent(out), target :: eval
      type(evaluator_t), intent(in), target :: eval_in1
      type(interaction_t), intent(in), target :: int_in2
      type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
      type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
      logical, intent(in), optional :: connections_are_resonant
      call evaluator_init_product_ii &
           (eval, eval_in1%int, int_in2, qn_mask_conn, qn_mask_rest, &
            connections_are_resonant)
    end subroutine evaluator_init_product_ei

    subroutine evaluator_init_product_ee &
         (eval, eval_in1, eval_in2, qn_mask_conn, qn_mask_rest, &
          connections_are_resonant)
      type(evaluator_t), intent(out), target :: eval
      type(evaluator_t), intent(in), target :: eval_in1, eval_in2
      type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
      type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
      logical, intent(in), optional :: connections_are_resonant
      call evaluator_init_product_ii &
           (eval, eval_in1%int, eval_in2%int, qn_mask_conn, qn_mask_rest, &
            connections_are_resonant)
    end subroutine evaluator_init_product_ee
```

7.7.5   Creating an evaluator: square

The generic initializer for an evaluator that squares a matrix element. Depending on the provided mask, we select the appropriate specific initializer for either diagonal or non-diagonal helicity density matrices.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_square
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_square (eval, int_in, qn_mask, &
         col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, dimension(:,:), intent(in), optional :: col_flow_index
      complex(default), dimension(:), intent(in), optional :: col_factor
      integer, dimension(:), intent(in), optional :: col_index_hi
      logical, intent(in), optional :: expand_color_flows
      integer, intent(in), optional :: nc
```

```
    if (all (quantum_numbers_mask_diagonal_helicity (qn_mask))) then
       call evaluator_init_square_diag (eval, int_in, qn_mask, &
            col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
    else
       call evaluator_init_square_nondiag (eval, int_in, qn_mask, &
            col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
    end if
  end subroutine evaluator_init_square
```

Color-summed squared matrix (diagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. The mask must be such that off-diagonal matrix elements are excluded.

If `color_flows` is set, the evaluator keeps color-flow entries separate and drops all interfering color structures. The color factors are set to unity in this case.

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to multiplication, with a few notable differences:

1. The connected particles are known, the correspondence is one-to-one. All particles are connected, and the mapping of indices is trivial, which simplifies the following steps.

2. `accumulate_connected_states`: The matrix of connected states encompasses all particles, but color indices are removed. However, ghost states are still kept separate from physical color states. No color-index reassignment is necessary.

3. The table of connections contains single index and quantum-number arrays instead of pairs of them. They are paired with themselves in all possible ways.

4. `make_squared_interaction`: Now apply the predefined quantum-numbers mask, which usually collects all color states (physical and ghosts), and possibly a helicity sum.

5. `make_pairing_array`: For each pair of input states, compute the color factor (including a potential ghost-parity sign) and store this in the pairing array together with the matrix-element indices for multiplication.

6. `record_links`: This is again trivial due to the one-to-one correspondence.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_square_diag
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_square_diag (eval, int_in, qn_mask, &
        col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)

      type(evaluator_t), intent(out), target :: eval
```

```
type(interaction_t), intent(in), target :: int_in
type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
integer, dimension(:,:), intent(in), optional :: col_flow_index
complex(default), dimension(:), intent(in), optional :: col_factor
integer, dimension(:), intent(in), optional :: col_index_hi
logical, intent(in), optional :: expand_color_flows
integer, intent(in), optional :: nc

integer :: n_in, n_vir, n_out, n_tot
type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial

type :: connection_table_t
  integer :: n_tot = 0
  integer :: n_me_conn = 0
  type(state_matrix_t) :: state
  type(index_map_t) :: index_conn
  type(connection_entry_t), dimension(:), allocatable :: entry
  type(index_map_t) :: index_result
end type connection_table_t
type(connection_table_t) :: connection_table

logical :: sum_colors
type(color_table_t) :: color_table

if (present (expand_color_flows)) then
   sum_colors = .not. expand_color_flows
else
   sum_colors = .true.
end if

if (sum_colors) then
   eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
else
   eval%type = EVAL_SQUARED_FLOWS
end if
eval%int_in1 => int_in

! print *, "Interaction square with color factors (diag)"  !!! Debugging
! print *, "Input interaction"                             !!! Debugging
! call interaction_write (int_in)                          !!! Debugging

n_in  = interaction_get_n_in  (int_in)
n_vir = interaction_get_n_vir (int_in)
n_out = interaction_get_n_out (int_in)
n_tot = interaction_get_n_tot (int_in)

allocate (qn_mask_initial (n_tot))
qn_mask_initial = interaction_get_mask (int_in)
call quantum_numbers_mask_set_color &
     (qn_mask_initial, sum_colors, mask_cg=.false.)
if (sum_colors) then
   call color_table_init &
        (color_table, interaction_get_state_matrix_ptr (int_in), n_tot)
   if (present (col_flow_index) .and. present (col_factor) &
```

847

```
                  .and. present (col_index_hi)) then
                call color_table_set_color_factors &
                     (color_table, col_flow_index, col_factor, col_index_hi)
            end if
          ! call color_table_write (color_table)     !!! Debugging
       end if

       call connection_table_init (connection_table, &
            interaction_get_state_matrix_ptr (int_in), &
            qn_mask_initial, qn_mask, n_tot)
       call connection_table_fill (connection_table, &
            interaction_get_state_matrix_ptr (int_in))
       call make_squared_interaction (eval%int, &
            n_in, n_vir, n_out, n_tot, &
            connection_table, sum_colors, qn_mask_initial .or. qn_mask)
       call make_pairing_array (eval%pairing_array, &
            interaction_get_n_matrix_elements (eval%int), &
            connection_table, sum_colors, color_table, n_in, n_tot, nc)
       call record_links (eval%int, int_in, n_tot)
       call connection_table_final (connection_table)
       ! print *, "Result evaluator:"     !!! Debugging
       ! call evaluator_write (eval)      !!! Debugging

  contains

    subroutine connection_table_init &
         (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
      type(connection_table_t), intent(out) :: connection_table
      type(state_matrix_t), intent(in), target :: state_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, intent(in) :: n_tot
      type(quantum_numbers_t), dimension(n_tot) :: qn
      type(state_iterator_t) :: it
      integer :: i, n_me_in, me_index_in
      integer :: me_index_conn, n_me_conn
      integer, dimension(1) :: me_count
      connection_table%n_tot = n_tot
      n_me_in = state_matrix_get_n_matrix_elements (state_in)
      call index_map_init (connection_table%index_conn, n_me_in)
      connection_table%index_conn = 0
      call state_matrix_init (connection_table%state, n_counters=1)
      call state_iterator_init (it, state_in)
      do while (state_iterator_is_valid (it))
         qn = state_iterator_get_quantum_numbers (it)
         if (all (quantum_numbers_are_physical (qn, qn_mask))) then
            call quantum_numbers_undefine (qn, qn_mask_in)
            me_index_in = state_iterator_get_me_index (it)
            call state_matrix_add_state (connection_table%state, qn, &
                 counter_index = 1, me_index = me_index_conn)
            call index_map_set_entry (connection_table%index_conn, &
                 me_index_in, me_index_conn)
         end if
         call state_iterator_advance (it)
```

```
      end do
      n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
      connection_table%n_me_conn = n_me_conn
      allocate (connection_table%entry (n_me_conn))
      call state_iterator_init (it, connection_table%state)
      do while (state_iterator_is_valid (it))
         i = state_iterator_get_me_index (it)
         me_count = state_iterator_get_me_count (it)
         call connection_entry_init (connection_table%entry(i), 1, 2, &
              state_iterator_get_quantum_numbers (it), me_count, (/n_tot/))
         call state_iterator_advance (it)
      end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
   type(connection_table_t), intent(inout) :: connection_table
   call state_matrix_final (connection_table%state)
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
   type(connection_table_t), intent(in) :: connection_table
   integer, intent(in), optional :: unit
   integer :: i
   integer :: u
   u = output_unit (unit)
   write (u, *) "Connection table:"
   call state_matrix_write (connection_table%state, unit)
   if (index_map_exists (connection_table%index_conn)) then
      write (u, *) "  Index mapping input => connection table:"
      do i = 1, size (connection_table%index_conn)
         write (u, *)  i, &
                index_map_get_entry (connection_table%index_conn, i)
      end do
   end if
   if (allocated (connection_table%entry)) then
      write (u, *) "  Connection table contents"
      do i = 1, size (connection_table%entry)
         call connection_entry_write (connection_table%entry(i), unit)
      end do
   end if
   if (index_map_exists (connection_table%index_result)) then
      write (u, *) "  Index mapping connection table => output"
      do i = 1, size (connection_table%index_result)
         write (u, *)  i, &
                index_map_get_entry (connection_table%index_result, i)
      end do
   end if
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
   type(connection_table_t), intent(inout) :: connection_table
   type(state_matrix_t), intent(in), target :: state
   integer :: index_in, index_conn, n_result_entries
   type(state_iterator_t) :: it
```

```
      integer :: k
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
          index_in = state_iterator_get_me_index (it)
          index_conn = &
              index_map_get_entry (connection_table%index_conn, index_in)
          if (index_conn /= 0) then
             call connection_entry_add_state &
                  (connection_table%entry(index_conn), &
                   index_in, &
                   state_iterator_get_quantum_numbers (it))
          end if
          call state_iterator_advance (it)
      end do
      n_result_entries = 0
      do k = 1, size (connection_table%entry)
          n_result_entries = &
              n_result_entries + connection_table%entry(k)%n_index(1) ** 2
      end do
      call index_map_init (connection_table%index_result, n_result_entries)
      connection_table%index_result = 0
    end subroutine connection_table_fill

    subroutine connection_entry_add_state (entry, index_in, qn_in)
      type(connection_entry_t), intent(inout) :: entry
      integer, intent(in) :: index_in
      type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
      integer :: c
      entry%count = entry%count + 1
      c = entry%count(1)
      call index_map_set_entry (entry%index_in(1), c, index_in)
      entry%qn_in_list(1)%qn(:,c) = qn_in
    end subroutine connection_entry_add_state

    subroutine make_squared_interaction (int, &
          n_in, n_vir, n_out, n_tot, &
          connection_table, sum_colors, qn_mask)
      type(interaction_t), intent(out), target :: int
      integer, intent(in) :: n_in, n_vir, n_out, n_tot
      type(connection_table_t), intent(inout), target :: connection_table
      logical, intent(in) :: sum_colors
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      type(connection_entry_t), pointer :: entry
      integer :: result_index, n_contrib
      integer :: i, m
      type(quantum_numbers_t), dimension(n_tot) :: qn
      call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
      m = 0
      do i = 1, connection_table%n_me_conn
          entry => connection_table%entry(i)
          qn = quantum_numbers_undefined (entry%qn_conn, qn_mask)
          if (.not. sum_colors)   call quantum_numbers_invert_color (qn(1:n_in))
          call interaction_add_state (int, qn, me_index = result_index)
          n_contrib = entry%n_index(1) ** 2
```

850

```
      connection_table%index_result%entry(m+1:m+n_contrib) = result_index
      m = m + n_contrib
   end do
   call interaction_freeze (int)
end subroutine make_squared_interaction

subroutine make_pairing_array (pa, &
      n_matrix_elements, connection_table, sum_colors, color_table, &
      n_in, n_tot, nc)
   type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
   integer, intent(in) :: n_matrix_elements
   type(connection_table_t), intent(in), target :: connection_table
   logical, intent(in) :: sum_colors
   type(color_table_t), intent(inout) :: color_table
   type(connection_entry_t), pointer :: entry
   integer, intent(in) :: n_in, n_tot
   integer, intent(in), optional :: nc
   integer, dimension(:), allocatable :: n_entries
   integer :: i, k, l, ks, ls, m, r
   integer :: color_multiplicity_in
   allocate (pa (n_matrix_elements))
   allocate (n_entries (n_matrix_elements))
   n_entries = 0
   do m = 1, size (connection_table%index_result)
      r = index_map_get_entry (connection_table%index_result, m)
      n_entries(r) = n_entries(r) + 1
   end do
   call pairing_array_init &
        (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
   m = 1
   n_entries = 0
   do i = 1, connection_table%n_me_conn
      entry => connection_table%entry(i)
      do k = 1, entry%n_index(1)
         if (sum_colors) then
            color_multiplicity_in = &
                 product (abs (quantum_numbers_get_color_type &
                             (entry%qn_in_list(1)%qn(:n_in, k))))
            do l = 1, entry%n_index(1)
               r = index_map_get_entry (connection_table%index_result, m)
               n_entries(r) = n_entries(r) + 1
               ks = index_map_get_entry (entry%index_in(1), k)
               ls = index_map_get_entry (entry%index_in(1), l)
               pa(r)%i1(n_entries(r)) = ks
               pa(r)%i2(n_entries(r)) = ls
               pa(r)%factor(n_entries(r)) = &
                    color_table_get_color_factor (color_table, ks, ls, nc) &
                    / color_multiplicity_in
               m = m + 1
            end do
         else
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
            ks = index_map_get_entry (entry%index_in(1), k)
```

```
                pa(r)%i1(n_entries(r)) = ks
                m = m + 1
             end if
          end do
       end do
    end subroutine make_pairing_array

    subroutine record_links (int, int_in, n_tot)
      type(interaction_t), intent(inout) :: int
      type(interaction_t), intent(in), target :: int_in
      integer, intent(in) :: n_tot
      integer, dimension(n_tot) :: map
      integer :: i
      do i = 1, n_tot
         call interaction_set_source_link (int, i, int_in, i)
      end do
      map = (/ (i, i = 1, n_tot) /)
      call interaction_transfer_relations (int_in, int, map)
    end subroutine record_links

  end subroutine evaluator_init_square_diag
```

Color-summed squared matrix (support nodiagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. Unless requested otherwise by the quantum-number mask, the result contains off-diagonal matrix elements. (The input interaction must be diagonal since it represents an amplitude, not a density matrix.)

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to the previous one, with some additional complications due to the necessity to loop over two helicity indices.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_square_nondiag
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_square_nondiag (eval, int_in, qn_mask, &
        col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)

      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      integer, dimension(:,:), intent(in), optional :: col_flow_index
      complex(default), dimension(:), intent(in), optional :: col_factor
      integer, dimension(:), intent(in), optional :: col_index_hi
      logical, intent(in), optional :: expand_color_flows
      integer, intent(in), optional :: nc

      integer :: n_in, n_vir, n_out, n_tot
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial
```

```
type :: connection_table_t
  integer :: n_tot = 0
  integer :: n_me_conn = 0
  type(state_matrix_t) :: state
  type(index_map2_t) :: index_conn
  type(connection_entry_t), dimension(:), allocatable :: entry
  type(index_map_t) :: index_result
end type connection_table_t
type(connection_table_t) :: connection_table

logical :: sum_colors
type(color_table_t) :: color_table

if (present (expand_color_flows)) then
   sum_colors = .not. expand_color_flows
else
   sum_colors = .true.
end if

if (sum_colors) then
   eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
else
   eval%type = EVAL_SQUARED_FLOWS
end if
eval%int_in1 => int_in

! print *, "Interaction square with color factors (nondiag)"  !!! Debugging
! print *, "Input interaction"                                 !!! Debugging
! call interaction_write (int_in)                              !!! Debugging
n_in  = interaction_get_n_in  (int_in)
n_vir = interaction_get_n_vir (int_in)
n_out = interaction_get_n_out (int_in)
n_tot = interaction_get_n_tot (int_in)

allocate (qn_mask_initial (n_tot))
qn_mask_initial = interaction_get_mask (int_in)
call quantum_numbers_mask_set_color &
     (qn_mask_initial, sum_colors, mask_cg=.false.)
if (sum_colors) then
   call color_table_init &
        (color_table, interaction_get_state_matrix_ptr (int_in), n_tot)
   if (present (col_flow_index) .and. present (col_factor) &
       .and. present (col_index_hi)) then
      call color_table_set_color_factors &
           (color_table, col_flow_index, col_factor, col_index_hi)
   end if
   ! call color_table_write (color_table)    !!! Debugging
end if

call connection_table_init (connection_table, &
     interaction_get_state_matrix_ptr (int_in), &
     qn_mask_initial, qn_mask, n_tot)
call connection_table_fill (connection_table, &
```

853

```
      interaction_get_state_matrix_ptr (int_in))
call make_squared_interaction (eval%int, &
     n_in, n_vir, n_out, n_tot, &
     connection_table, sum_colors, qn_mask_initial .or. qn_mask)
! call connection_table_write (connection_table)     !!! Debugging
call make_pairing_array (eval%pairing_array, &
     interaction_get_n_matrix_elements (eval%int), &
     connection_table, sum_colors, color_table, n_in, n_tot, nc)
call record_links (eval%int, int_in, n_tot)
call connection_table_final (connection_table)

! print *, "Result evaluator:"     !!! Debugging
! call evaluator_write (eval)       !!! Debugging

contains

  subroutine connection_table_init &
       (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
    type(connection_table_t), intent(out) :: connection_table
    type(state_matrix_t), intent(in), target :: state_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    integer, intent(in) :: n_tot
    type(quantum_numbers_t), dimension(n_tot) :: qn1, qn2, qn
    type(state_iterator_t) :: it1, it2, it
    integer :: i, n_me_in, me_index_in1, me_index_in2
    integer :: me_index_conn, n_me_conn
    integer, dimension(1) :: me_count
    connection_table%n_tot = n_tot
    n_me_in = state_matrix_get_n_matrix_elements (state_in)
    call index_map2_init (connection_table%index_conn, n_me_in)
    connection_table%index_conn = 0
    call state_matrix_init (connection_table%state, n_counters=1)
    call state_iterator_init (it1, state_in)
    do while (state_iterator_is_valid (it1))
       qn1 = state_iterator_get_quantum_numbers (it1)
       me_index_in1 = state_iterator_get_me_index (it1)
       call state_iterator_init (it2, state_in)
       do while (state_iterator_is_valid (it2))
          qn2 = state_iterator_get_quantum_numbers (it2)
          if (all (quantum_numbers_are_compatible (qn1, qn2, qn_mask))) then
             qn = qn1 .merge. qn2
             call quantum_numbers_undefine (qn, qn_mask_in)
             me_index_in2 = state_iterator_get_me_index (it2)
             call state_matrix_add_state (connection_table%state, qn, &
                  counter_index = 1, me_index = me_index_conn)
             call index_map2_set_entry (connection_table%index_conn, &
                  me_index_in1, me_index_in2, me_index_conn)
          end if
          call state_iterator_advance (it2)
       end do
       call state_iterator_advance (it1)
    end do
    n_me_conn = state_matrix_get_n_matrix_elements (connection_table%state)
```

```fortran
      connection_table%n_me_conn = n_me_conn
      allocate (connection_table%entry (n_me_conn))
      call state_iterator_init (it, connection_table%state)
      do while (state_iterator_is_valid (it))
         i = state_iterator_get_me_index (it)
         me_count = state_iterator_get_me_count (it)
         call connection_entry_init (connection_table%entry(i), 1, 2, &
               state_iterator_get_quantum_numbers (it), me_count, (/n_tot/))
         call state_iterator_advance (it)
      end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
   type(connection_table_t), intent(inout) :: connection_table
   call state_matrix_final (connection_table%state)
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
   type(connection_table_t), intent(in) :: connection_table
   integer, intent(in), optional :: unit
   integer :: i, j
   integer :: u
   u = output_unit (unit)
   write (u, *) "Connection table:"
   call state_matrix_write (connection_table%state, unit)
   if (index_map2_exists (connection_table%index_conn)) then
      write (u, *) "  Index mapping input => connection table:"
      do i = 1, size (connection_table%index_conn)
         do j = 1, size (connection_table%index_conn)
            write (u, *)  i, j, &
                  index_map2_get_entry (connection_table%index_conn, i, j)
         end do
      end do
   end if
   if (allocated (connection_table%entry)) then
      write (u, *) "  Connection table contents"
      do i = 1, size (connection_table%entry)
         call connection_entry_write (connection_table%entry(i), unit)
      end do
   end if
   if (index_map_exists (connection_table%index_result)) then
      write (u, *) "  Index mapping connection table => output"
      do i = 1, size (connection_table%index_result)
         write (u, *)  i, &
               index_map_get_entry (connection_table%index_result, i)
      end do
   end if
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
   type(connection_table_t), intent(inout), target :: connection_table
   type(state_matrix_t), intent(in), target :: state
   integer :: index1_in, index2_in, index_conn, n_result_entries
   type(state_iterator_t) :: it1, it2
```

855

```fortran
    integer :: k
  call state_iterator_init (it1, state)
  do while (state_iterator_is_valid (it1))
     index1_in = state_iterator_get_me_index (it1)
     call state_iterator_init (it2, state)
     do while (state_iterator_is_valid (it2))
        index2_in = state_iterator_get_me_index (it2)
        index_conn = index_map2_get_entry &
                      (connection_table%index_conn, index1_in, index2_in)
        if (index_conn /= 0) then
           call connection_entry_add_state &
                (connection_table%entry(index_conn), &
                 index1_in, index2_in, &
                 state_iterator_get_quantum_numbers (it1) &
                 .merge. &
                 state_iterator_get_quantum_numbers (it2))
        end if
        call state_iterator_advance (it2)
     end do
     call state_iterator_advance (it1)
  end do
  n_result_entries = 0
  do k = 1, size (connection_table%entry)
     n_result_entries = &
          n_result_entries + connection_table%entry(k)%n_index(1)
  end do
  call index_map_init (connection_table%index_result, n_result_entries)
  connection_table%index_result = 0
end subroutine connection_table_fill

subroutine connection_entry_add_state (entry, index1_in, index2_in, qn_in)
  type(connection_entry_t), intent(inout) :: entry
  integer, intent(in) :: index1_in, index2_in
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
  integer :: c
  entry%count = entry%count + 1
  c = entry%count(1)
  call index_map_set_entry (entry%index_in(1), c, index1_in)
  call index_map_set_entry (entry%index_in(2), c, index2_in)
  entry%qn_in_list(1)%qn(:,c) = qn_in
end subroutine connection_entry_add_state

subroutine make_squared_interaction (int, &
     n_in, n_vir, n_out, n_tot, &
     connection_table, sum_colors, qn_mask)
  type(interaction_t), intent(out), target :: int
  integer, intent(in) :: n_in, n_vir, n_out, n_tot
  type(connection_table_t), intent(inout), target :: connection_table
  logical, intent(in) :: sum_colors
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  type(connection_entry_t), pointer :: entry
  integer :: result_index
  integer :: i, k, m
  type(quantum_numbers_t), dimension(n_tot) :: qn
```

```
    call interaction_init (eval%int, n_in, n_vir, n_out, mask=qn_mask)
    m = 0
    do i = 1, connection_table%n_me_conn
        entry => connection_table%entry(i)
        do k = 1, size (entry%qn_in_list(1)%qn, 2)
            qn = quantum_numbers_undefined &
                    (entry%qn_in_list(1)%qn(:,k), qn_mask)
            if (.not. sum_colors) &
                    call quantum_numbers_invert_color (qn(1:n_in))
            call interaction_add_state (int, qn, me_index = result_index)
            call index_map_set_entry (connection_table%index_result, m + 1, &
                result_index)
            m = m + 1
        end do
    end do
    call interaction_freeze (int)
end subroutine make_squared_interaction

subroutine make_pairing_array (pa, &
        n_matrix_elements, connection_table, sum_colors, color_table, &
        n_in, n_tot, nc)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, intent(in) :: n_matrix_elements
    type(connection_table_t), intent(in), target :: connection_table
    logical, intent(in) :: sum_colors
    type(color_table_t), intent(inout) :: color_table
    type(connection_entry_t), pointer :: entry
    integer, intent(in) :: n_in, n_tot
    integer, intent(in), optional :: nc
    integer, dimension(:), allocatable :: n_entries
    integer :: i, k, k1s, k2s, m, r
    integer :: color_multiplicity_in
    allocate (pa (n_matrix_elements))
    allocate (n_entries (n_matrix_elements))
    n_entries = 0
    do m = 1, size (connection_table%index_result)
        r = index_map_get_entry (connection_table%index_result, m)
        n_entries(r) = n_entries(r) + 1
    end do
    call pairing_array_init &
        (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
    m = 1
    n_entries = 0
    do i = 1, connection_table%n_me_conn
        entry => connection_table%entry(i)
        do k = 1, entry%n_index(1)
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
            if (sum_colors) then
                k1s = index_map_get_entry (entry%index_in(1), k)
                k2s = index_map_get_entry (entry%index_in(2), k)
                pa(r)%i1(n_entries(r)) = k1s
                pa(r)%i2(n_entries(r)) = k2s
                color_multiplicity_in = &
```

```
                     product (abs (quantum_numbers_get_color_type &
                                    (entry%qn_in_list(1)%qn(:n_in, k))))
               pa(r)%factor(n_entries(r)) = &
                     color_table_get_color_factor (color_table, k1s, k2s, nc) &
                     / color_multiplicity_in
           else
               k1s = index_map_get_entry (entry%index_in(1), k)
               pa(r)%i1(n_entries(r)) = k1s
           end if
           m = m + 1
        end do
     end do
   end subroutine make_pairing_array

   subroutine record_links (int, int_in, n_tot)
     type(interaction_t), intent(inout) :: int
     type(interaction_t), intent(in), target :: int_in
     integer, intent(in) :: n_tot
     integer, dimension(n_tot) :: map
     integer :: i
     do i = 1, n_tot
        call interaction_set_source_link (int, i, int_in, i)
     end do
     map = (/ (i, i = 1, n_tot) /)
     call interaction_transfer_relations (int_in, int, map)
   end subroutine record_links

 end subroutine evaluator_init_square_nondiag
```

Copy with additional contracted color states

This evaluator involves no square or multiplication, its matrix elements are
just copies of the (single) input interaction. However, the state matrix of the
interaction contains additional states that have color indices contracted. This
is used for copies of the beam or structure-function interactions that need to
match the hard interaction also in the case where its color indices coincide.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_color_contractions
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_color_contractions (eval, int_in)
      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int_in
      integer :: n_in, n_vir, n_out, n_tot
      type(state_matrix_t) :: state_with_contractions
      integer, dimension(:), allocatable :: me_index
      integer, dimension(:), allocatable :: result_index
      eval%type = EVAL_COLOR_CONTRACTION
      eval%int_in1 => int_in
      ! print *, "Interaction with additional color contractions"  !!! Debugging
      ! print *, "Input interaction"                               !!! Debugging
      ! call interaction_write (int_in)                            !!! Debugging
      n_in  = interaction_get_n_in  (int_in)
```

```
      n_vir = interaction_get_n_vir (int_in)
      n_out = interaction_get_n_out (int_in)
      n_tot = interaction_get_n_tot (int_in)
      state_with_contractions = interaction_get_state_matrix_ptr (int_in)
      call state_matrix_add_color_contractions (state_with_contractions)
      call make_contracted_interaction (eval%int, &
           me_index, result_index, &
           n_in, n_vir, n_out, n_tot, &
           state_with_contractions, interaction_get_mask (int_in))
      call make_pairing_array (eval%pairing_array, me_index, result_index)
      call record_links (eval%int, int_in, n_tot)
      call state_matrix_final (state_with_contractions)
      ! print *, "Result evaluator:"      !!! Debugging
      ! call evaluator_write (eval)       !!! Debugging

  contains

    subroutine make_contracted_interaction (int, &
         me_index, result_index, &
         n_in, n_vir, n_out, n_tot, state, qn_mask)
      type(interaction_t), intent(out), target :: int
      integer, dimension(:), intent(out), allocatable :: me_index
      integer, dimension(:), intent(out), allocatable :: result_index
      integer, intent(in) :: n_in, n_vir, n_out, n_tot
      type(state_matrix_t), intent(in) :: state
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
      type(state_iterator_t) :: it
      integer :: n_me, i
      type(quantum_numbers_t), dimension(n_tot) :: qn
      call interaction_init (int, n_in, n_vir, n_out, mask=qn_mask)
      n_me = state_matrix_get_n_leaves (state)
      allocate (me_index (n_me))
      allocate (result_index (n_me))
      call state_iterator_init (it, state)
      i = 0
      do while (state_iterator_is_valid (it))
         i = i + 1
         me_index(i) = state_iterator_get_me_index (it)
         qn = state_iterator_get_quantum_numbers (it)
         call interaction_add_state (int, qn, me_index = result_index(i))
         call state_iterator_advance (it)
      end do
      call interaction_freeze (int)
    end subroutine make_contracted_interaction

    subroutine make_pairing_array (pa, me_index, result_index)
      type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
      integer, dimension(:), intent(in) :: me_index, result_index
      integer, dimension(:), allocatable :: n_entries
      integer :: n_matrix_elements, r, i
      n_matrix_elements = size (me_index)
      allocate (pa (n_matrix_elements))
      allocate (n_entries (n_matrix_elements))
      n_entries = 1
```

```
      call pairing_array_init &
          (pa, n_entries, has_i2=.false., has_factor=.false.)
      do i = 1, n_matrix_elements
         r = result_index(i)
         pa(r)%i1(1) = me_index(i)
      end do
    end subroutine make_pairing_array

    subroutine record_links (int, int_in, n_tot)
      type(interaction_t), intent(inout) :: int
      type(interaction_t), intent(in), target :: int_in
      integer, intent(in) :: n_tot
      integer, dimension(n_tot) :: map
      integer :: i
      do i = 1, n_tot
         call interaction_set_source_link (int, i, int_in, i)
      end do
      map = (/ (i, i = 1, n_tot) /)
      call interaction_transfer_relations (int_in, int, map)
    end subroutine record_links

  end subroutine evaluator_init_color_contractions
```

Auxiliary procedure for initialization

This will become a standard procedure in F2008. The result is true if the number of true values in the mask is odd. We use the function for determining the ghost parity of a quantum-number array.

⟨*Evaluators: procedures*⟩+≡
```
    function parity (mask)
      logical :: parity
      logical, dimension(:) :: mask
      integer :: i
      parity = .false.
      do i = 1, size (mask)
         if (mask(i))  parity = .not. parity
      end do
    end function parity
```

7.7.6   Accessing contents

Return the interaction component, as a pointer to avoid any copying.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_int_ptr
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_int_ptr (eval) result (int)
      type(interaction_t), pointer :: int
      type(evaluator_t), intent(in), target :: eval
      int => eval%int
    end function evaluator_get_int_ptr
```

7.7.7 Inherited procedures

Return true if the state matrix within the interaction is empty.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_is_empty
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_is_empty (eval) result (flag)
      logical :: flag
      type(evaluator_t), intent(in) :: eval
      flag = interaction_is_empty (eval%int)
    end function evaluator_is_empty
```

Return the integer tag.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_tag
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_tag (eval) result (tag)
      integer :: tag
      type(evaluator_t), intent(in) :: eval
      tag = interaction_get_tag (eval%int)
    end function evaluator_get_tag
```

Get the norm of the state matrix (if the norm has been taken out, otherwise this would be unity).

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_norm
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_norm (eval) result (norm)
      real(default) :: norm
      type(evaluator_t), intent(in) :: eval
      norm = interaction_get_norm (eval%int)
    end function evaluator_get_norm
```

Return the number of particles.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_n_tot
    public :: evaluator_get_n_in
    public :: evaluator_get_n_vir
    public :: evaluator_get_n_out
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_n_tot (eval) result (n_tot)
      integer :: n_tot
      type(evaluator_t), intent(in) :: eval
      n_tot = interaction_get_n_tot (eval%int)
    end function evaluator_get_n_tot

    function evaluator_get_n_in (eval) result (n_in)
      integer :: n_in
      type(evaluator_t), intent(in) :: eval
      n_in = interaction_get_n_in (eval%int)
```

```
    end function evaluator_get_n_in

    function evaluator_get_n_vir (eval) result (n_vir)
      integer :: n_vir
      type(evaluator_t), intent(in) :: eval
      n_vir = interaction_get_n_vir (eval%int)
    end function evaluator_get_n_vir

    function evaluator_get_n_out (eval) result (n_out)
      integer :: n_out
      type(evaluator_t), intent(in) :: eval
      n_out = interaction_get_n_out (eval%int)
    end function evaluator_get_n_out
```

Return a specific matrix element.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_matrix_element
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_matrix_element (eval, i) result (value)
      complex(default) :: value
      type(evaluator_t), intent(in) :: eval
      integer :: i
      value = interaction_get_matrix_element (eval%int, i)
    end function evaluator_get_matrix_element
```

Sum all matrix element values.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_sum
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_sum (eval) result (value)
      complex(default) :: value
      type(evaluator_t), intent(in) :: eval
      value = interaction_sum (eval%int)
    end function evaluator_sum
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is
reflected in the state-matrix norm.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_normalize_by_trace
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_normalize_by_trace (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_normalize_by_trace (eval%int)
    end subroutine evaluator_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_normalize_by_max
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_normalize_by_max (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_normalize_by_max (eval%int)
    end subroutine evaluator_normalize_by_max
```

Append color-contracted states. Matrix element array and multiplication table
are unaffected by this.
⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_add_color_contractions
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_add_color_contractions (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_add_color_contractions (eval%int)
    end subroutine evaluator_add_color_contractions
```

Return the quantum-numbers mask of the enclosed interaction.
⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_mask
```

⟨*Evaluators: procedures*⟩+≡
```
    function evaluator_get_mask (eval) result (mask)
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
      type(evaluator_t), intent(in), target :: eval
      allocate (mask (interaction_get_n_tot (eval%int)))
      mask = interaction_get_mask (eval%int)
    end function evaluator_get_mask
```

Extend the linking of interactions to evaluators.
⟨*Evaluators: public*⟩+≡
```
    public :: interaction_set_source_link
    public :: evaluator_set_source_link
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface interaction_set_source_link
       module procedure interaction_set_source_link_eval
    end interface
    interface evaluator_set_source_link
       module procedure evaluator_set_source_link_int
       module procedure evaluator_set_source_link_eval
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine interaction_set_source_link_eval (int, i, eval1, i1)
      type(interaction_t), intent(inout) :: int
      type(evaluator_t), intent(in), target :: eval1
      integer, intent(in) :: i, i1
      call interaction_set_source_link (int, i, eval1%int, i1)
    end subroutine interaction_set_source_link_eval

    subroutine evaluator_set_source_link_int (eval, i, int1, i1)
      type(evaluator_t), intent(inout) :: eval
      type(interaction_t), intent(in), target :: int1
```

863

```
      integer, intent(in) :: i, i1
      call interaction_set_source_link (eval%int, i, int1, i1)
    end subroutine evaluator_set_source_link_int

    subroutine evaluator_set_source_link_eval (eval, i, eval1, i1)
      type(evaluator_t), intent(inout) :: eval
      type(evaluator_t), intent(in), target :: eval1
      integer, intent(in) :: i, i1
      call interaction_set_source_link (eval%int, i, eval1%int, i1)
    end subroutine evaluator_set_source_link_eval
```

Receive from / send momenta to the linked interactions.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_receive_momenta
    public :: evaluator_send_momenta
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_receive_momenta (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_receive_momenta (eval%int)
    end subroutine evaluator_receive_momenta

    subroutine evaluator_send_momenta (eval)
      type(evaluator_t), intent(in) :: eval
      call interaction_send_momenta (eval%int)
    end subroutine evaluator_send_momenta
```

Reassign external source links from one to another.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_reassign_links
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface evaluator_reassign_links
       module procedure evaluator_reassign_links_eval
       module procedure evaluator_reassign_links_int
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_reassign_links_eval (eval, eval_src, eval_target)
      type(evaluator_t), intent(inout) :: eval
      type(evaluator_t), intent(in) :: eval_src
      type(evaluator_t), intent(in), target :: eval_target
      if (associated (eval%int_in1)) then
         if (interaction_get_tag (eval%int_in1) &
              == interaction_get_tag (eval_src%int)) then
            eval%int_in1 => eval_target%int
         end if
      end if
      if (associated (eval%int_in2)) then
         if (interaction_get_tag (eval%int_in2) &
              == interaction_get_tag (eval_src%int)) then
            eval%int_in2 => eval_target%int
         end if
```

```
      end if
      call interaction_reassign_links (eval%int, eval_src%int, eval_target%int)
    end subroutine evaluator_reassign_links_eval

    subroutine evaluator_reassign_links_int (eval, int_src, int_target)
      type(evaluator_t), intent(inout) :: eval
      type(interaction_t), intent(in) :: int_src
      type(interaction_t), intent(in), target :: int_target
      if (associated (eval%int_in1)) then
         if (interaction_get_tag (eval%int_in1) &
              == interaction_get_tag (int_src)) then
            eval%int_in1 => int_target
         end if
      end if
      if (associated (eval%int_in2)) then
         if (interaction_get_tag (eval%int_in2) &
              == interaction_get_tag (int_src)) then
            eval%int_in2 => int_target
         end if
      end if
      call interaction_reassign_links (eval%int, int_src, int_target)
    end subroutine evaluator_reassign_links_int
```

Return flavor, momentum, and position of the first unstable particle present in
the interaction.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_get_unstable_particle
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_get_unstable_particle (eval, flv, p, i)
      type(evaluator_t), intent(in) :: eval
      type(flavor_t), intent(out) :: flv
      type(vector4_t), intent(out) :: p
      integer, intent(out) :: i
      call interaction_get_unstable_particle (eval%int, flv, p, i)
    end subroutine evaluator_get_unstable_particle
```

7.7.8 Deleting the evaluator

Only the interaction component needs finalization.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_final
```

⟨*Evaluators: procedures*⟩+≡
```
    elemental subroutine evaluator_final (eval)
      type(evaluator_t), intent(inout) :: eval
      call interaction_final (eval%int)
    end subroutine evaluator_final
```

7.7.9 Creating an evaluator: identity

The identity evaluator creates a copy of the first input evaluator; the second input is not used.

All particles link back to the input evaluatorand the internal relations are copied. As evaluation does take a shortcut by cloning the matrix elements, the pairing array is not used and does not have to be set up.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_init_identity
```

⟨*Evaluators: interfaces*⟩+≡
```
    interface evaluator_init_identity
       module procedure evaluator_init_identity_i
       module procedure evaluator_init_identity_e
    end interface
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_init_identity_i (eval, int)
      type(evaluator_t), intent(out), target :: eval
      type(interaction_t), intent(in), target :: int
      integer :: n_in, n_out, n_vir, n_tot
      integer :: i
      integer, dimension(:), allocatable :: map
      type(state_matrix_t), pointer :: state
      type(state_iterator_t) :: it

      eval%type = EVAL_IDENTITY
      eval%int_in1 => int
      nullify (eval%int_in2)
      n_in = interaction_get_n_in (int)
      n_out = interaction_get_n_out (int)
      n_vir = interaction_get_n_vir (int)
      n_tot = interaction_get_n_tot (int)
      call interaction_init (eval%int, n_in, n_vir, n_out, &
         mask=interaction_get_mask (int), &
         resonant=interaction_get_resonance_flags (int))
      do i = 1, n_tot
         call interaction_set_source_link (eval%int, i, int, i)
      end do
      allocate (map(n_tot))
      map = (/(i, i = 1, n_tot)/)
      call interaction_transfer_relations (int, eval%int, map)
      state => interaction_get_state_matrix_ptr (int)
      call state_iterator_init (it, state)
      do while (state_iterator_is_valid (it))
         call interaction_add_state (eval%int, &
            state_iterator_get_quantum_numbers (it), &
            state_iterator_get_me_index (it))
         call state_iterator_advance (it)
      end do
      call interaction_freeze (eval%int)

    end subroutine evaluator_init_identity_i
```

```
      subroutine evaluator_init_identity_e (eval, eval1)
        type(evaluator_t), intent(out), target :: eval
        type(evaluator_t), intent(in), target :: eval1
        call evaluator_init_identity_i (eval, eval1%int)
      end subroutine evaluator_init_identity_e
```

7.7.10 Creating an evaluator: quantum number sum

This evaluator operates on the diagonal of a density matrix and sums over the
quantum numbers specified by the mask. The optional argument `drop` allows
to drop a particle from the resulting density matrix. The handling of virtuals
is not completely sane, especially in connection with dropping particles.

When summing over matrix element entries, we keep the separation into
entries and normalization (in the corresponding evaluation routine below).

⟨*Evaluators: public*⟩+≡
```
      public :: evaluator_init_qn_sum
```

⟨*Evaluators: interfaces*⟩+≡
```
      interface evaluator_init_qn_sum
         module procedure evaluator_init_qn_sum_i
         module procedure evaluator_init_qn_sum_e
      end interface
```

⟨*Evaluators: procedures*⟩+≡
```
      subroutine evaluator_init_qn_sum_i (eval, int, qn_mask, drop)
        type(evaluator_t), intent(out), target :: eval
        type(interaction_t), target, intent(in) :: int
        type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
        logical, intent(in), optional, dimension(:) :: drop
        type(state_iterator_t) :: it_old, it_new
        integer, dimension(:,:), allocatable :: pairing_proto
        integer, dimension(:), allocatable :: pairing_sizes
        integer, dimension(:), allocatable :: map
        integer :: n_in, n_out, n_vir, n_tot, n_me_old, n_me_new
        integer :: i, j
        type(state_matrix_t), pointer :: state_new, state_old
        type(quantum_numbers_t), dimension(:), allocatable :: qn
        logical :: matched
        logical, dimension(size (qn_mask)) :: dropped
        integer :: ndropped
        integer, dimension(:), allocatable :: inotdropped
        type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
        logical, dimension(:), allocatable :: resonant

        eval%type = EVAL_QN_SUM
        eval%int_in1 => int
        nullify (eval%int_in2)
        if (present (drop)) then
           dropped = drop
        else
           dropped = .false.
        end if
        ndropped = count (dropped)
```

867

```
n_in = interaction_get_n_in (int)
n_out = interaction_get_n_out (int) - ndropped
n_vir = interaction_get_n_vir (int)
n_tot = interaction_get_n_tot (int) - ndropped

allocate (inotdropped (n_tot))
i = 1
do j = 1, n_tot + ndropped
   if (dropped (j)) cycle
   inotdropped(i) = j
   i = i + 1
end do

allocate (mask(n_tot + ndropped))
mask = interaction_get_mask (int)
allocate (resonant(n_tot + ndropped))
resonant = interaction_get_resonance_flags (int)
call interaction_init (eval%int, n_in, n_vir, n_out, &
   mask = mask(inotdropped) .or. qn_mask(inotdropped), &
   resonant = resonant(inotdropped))
i = 1
do j = 1, n_tot + ndropped
   if (dropped(j)) cycle
   call interaction_set_source_link (eval%int, i, int, j)
   i = i + 1
end do
allocate (map(n_tot + ndropped))
i = 1
do j = 1, n_tot + ndropped
   if (dropped (j)) then
      map(j) = 0
   else
      map(j) = i
      i = i + 1
   end if
end do
call interaction_transfer_relations (int, eval%int, map)

n_me_old = interaction_get_n_matrix_elements (int)
allocate (pairing_proto(n_me_old, n_me_old))
allocate (pairing_sizes(n_me_old))
pairing_sizes = 0
state_old => interaction_get_state_matrix_ptr (int)
state_new => interaction_get_state_matrix_ptr (eval%int)
call state_iterator_init (it_old, state_old)
allocate (qn(n_tot + ndropped))
do while (state_iterator_is_valid (it_old))
   qn = state_iterator_get_quantum_numbers (it_old)
   if (.not. all (quantum_numbers_are_diagonal (qn))) then
      call state_iterator_advance (it_old)
      cycle
   end if
   matched = .false.
```

```
      call state_iterator_init (it_new, state_new)
      if (interaction_get_n_matrix_elements (eval%int) > 0) then
          do while (state_iterator_is_valid (it_new))
              if (all (qn(inotdropped) .match. &
                  state_iterator_get_quantum_numbers (it_new))) &
              then
                  matched = .true.
                  i = state_iterator_get_me_index (it_new)
                  exit
              end if
              call state_iterator_advance (it_new)
          end do
      end if
      if (.not. matched) then
          call interaction_add_state (eval%int, qn(inotdropped))
          i = interaction_get_n_matrix_elements (eval%int)
      end if
      pairing_sizes(i) = pairing_sizes(i) + 1
      pairing_proto(pairing_sizes(i), i) = &
          state_iterator_get_me_index (it_old)
      ! print *, &                           !!! Debugging
      !     i, pairing_sizes(i), state_iterator_get_me_index (it_old)
      call state_iterator_advance (it_old)
  end do
  call interaction_freeze (eval%int)

  n_me_new = interaction_get_n_matrix_elements (eval%int)
  allocate (eval%pairing_array(n_me_new))
  do i = 1, n_me_new
      ! print *, i, pairing_sizes(i)        !!! Debugging
      call pairing_array_init (eval%pairing_array(i), &
          pairing_sizes(i), .false., .false.)
      eval%pairing_array(i)%i1 = pairing_proto(:pairing_sizes(i), i)
  end do

end subroutine evaluator_init_qn_sum_i

subroutine evaluator_init_qn_sum_e (eval, eval1, qn_mask, drop)
  type(evaluator_t), intent(out) :: eval
  type(evaluator_t), intent(in), target :: eval1
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  logical, dimension(:), optional, intent(in) :: drop
  call evaluator_init_qn_sum_i (eval, eval1%int, qn_mask, drop)
end subroutine
```

7.7.11 Evaluation

When the input interactions (which are pointed to in the pairings stored within the evaluator) are filled with values, we can activate the evaluator, i.e., calculate the result values which are stored in the interaction.

The evaluation of matrix elements can be done in parallel. A `forall` construct is not appropriate, however. We would need `do concurrent` here. Nev-

ertheless, the evaluation functions are marked as `pure`.

⟨*Evaluators: evaluator: TBP*⟩+≡
```
    procedure :: evaluate => evaluator_evaluate
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_evaluate (eval)
      class(evaluator_t), intent(inout), target :: eval
      integer :: i
      select case (eval%type)
      case (EVAL_PRODUCT)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_product (eval%int, i, &
                 eval%int_in1, eval%int_in2, &
                 eval%pairing_array(i)%i1, eval%pairing_array(i)%i2)
         end do
      case (EVAL_SQUARE_WITH_COLOR_FACTORS)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_product_cf (eval%int, i, &
                 eval%int_in1, eval%int_in1, &
                 eval%pairing_array(i)%i1, eval%pairing_array(i)%i2, &
                 eval%pairing_array(i)%factor)
         end do
      case (EVAL_SQUARED_FLOWS)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_square_c (eval%int, i, &
                 eval%int_in1, &
                 eval%pairing_array(i)%i1)
         end do
      case (EVAL_COLOR_CONTRACTION)
         do i = 1, size(eval%pairing_array)
            call interaction_evaluate_sum (eval%int, i, &
                 eval%int_in1, &
                 eval%pairing_array(i)%i1)
         end do
      case (EVAL_IDENTITY)
         call interaction_set_matrix_element (eval%int, eval%int_in1)
      case (EVAL_QN_SUM)
         do i = 1, size (eval%pairing_array)
            call interaction_evaluate_me_sum (eval%int, i, &
               eval%int_in1, eval%pairing_array(i)%i1)
            call interaction_set_norm &
                 (eval%int, interaction_get_norm (eval%int_in1))
         end do
      end select
    end subroutine evaluator_evaluate
```

7.7.12 Unit tests

Test: Create two interactions. The interactions are twofold connected. The first connection has a helicity index that is kept, the second connection has a helicity index that is summed over. Concatenate the interactions in an evaluator, which thus contains a result interaction. Fill the input interactions with values,

activate the evaluator and print the result.

⟨*Evaluators: public*⟩+≡
```
    public :: evaluator_test
```

⟨*Evaluators: procedures*⟩+≡
```
    subroutine evaluator_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Evaluators: execute tests⟩
    end subroutine evaluator_test
```

⟨*Evaluators: execute tests*⟩≡
```
    call test (evaluator_1, "evaluator_1", &
         "check evaluators (1)", &
         u, results)
```

⟨*Evaluators: tests*⟩≡
```
    subroutine evaluator_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(interaction_t), target :: int_qqtt, int_tbw, int1, int2
      type(flavor_t), dimension(:), allocatable :: flv
      type(color_t), dimension(:), allocatable :: col
      type(helicity_t), dimension(:), allocatable :: hel
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      integer :: f, c, h1, h2, h3
      type(vector4_t), dimension(4) :: p
      type(vector4_t), dimension(2) :: q
      type(quantum_numbers_mask_t) :: qn_mask_conn
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask2
      type(evaluator_t), target :: eval, eval2, eval3

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("SM"), &
           var_str ("SM.mdl"), os_data, model)

      write (u, "(A)")   "*** Evaluator for matrix product"
      write (u, "(A)")   "***   Construct interaction for qq -> tt"
      write (u, "(A)")
      call interaction_init (int_qqtt, 2, 0, 2, set_relations=.true.)
      allocate (flv (4), col (4), hel (4), qn (4))
      allocate (qn_mask2 (4))
      do c = 1, 2
         select case (c)
         case (1)
            call color_init_col_acl (col, [1, 0, 1, 0], [0, 2, 0, 2])
         case (2)
            call color_init_col_acl (col, [1, 0, 2, 0], [0, 1, 0, 2])
         end select
         do f = 1, 2
            call flavor_init (flv, [f, -f, 6, -6], model)
            do h1 = -1, 1, 2
```

```
               call helicity_init (hel(3), h1)
            do h2 = -1, 1, 2
                call helicity_init (hel(4), h2)
                call quantum_numbers_init (qn, flv, col, hel)
                call interaction_add_state (int_qqtt, qn)
            end do
         end do
      end do
   end do
   call interaction_freeze (int_qqtt)
   deallocate (flv, col, hel, qn)
   write (u, "(A)")  "***   Construct interaction for t -> bW"
   call interaction_init (int_tbw, 1, 0, 2, set_relations=.true.)
   allocate (flv (3), col (3), hel (3), qn (3))
   call flavor_init (flv, [6, 5, 24], model)
   call color_init_col_acl (col, [1, 1, 0], [0, 0, 0])
   do h1 = -1, 1, 2
      call helicity_init (hel(1), h1)
      do h2 = -1, 1, 2
          call helicity_init (hel(2), h2)
          do h3 = -1, 1
              call helicity_init (hel(3), h3)
              call quantum_numbers_init (qn, flv, col, hel)
              call interaction_add_state (int_tbw, qn)
          end do
      end do
   end do
   call interaction_freeze (int_tbw)
   deallocate (flv, col, hel, qn)
   write (u, "(A)")  "***   Link interactions"
   call interaction_set_source_link (int_tbw, 1, int_qqtt, 3)
   qn_mask_conn = new_quantum_numbers_mask (.false.,.false.,.true.)
   write (u, "(A)")
   write (u, "(A)")  "***   Show input"
   call interaction_write (int_qqtt, unit = u)
   write (u, "(A)")
   call interaction_write (int_tbw, unit = u)
   write (u, "(A)")
   write (u, "(A)")  "***   Evaluate product"
   call evaluator_init_product &
        (eval, int_qqtt, int_tbw, qn_mask_conn)
   call eval%write (unit = u)

    call interaction_init (int1, 2, 0, 2, set_relations=.true.)
    call interaction_init (int2, 1, 0, 2, set_relations=.true.)
    p(1) = vector4_moving (1000._default, 1000._default, 3)
    p(2) = vector4_moving (200._default, 200._default, 2)
    p(3) = vector4_moving (100._default, 200._default, 1)
    p(4) = p(1) - p(2) - p(3)
    call interaction_set_momenta (int1, p)
    q(1) = vector4_moving (50._default,-50._default, 3)
    q(2) = p(2) + p(4) - q(1)
    call interaction_set_momenta (int2, q, outgoing=.true.)
    call interaction_set_matrix_element &
```

```
      (int1, (/(2._default,0._default), (4._default,1._default), (-3._default,0._default)/))
call interaction_set_matrix_element &
      (int2, (/(-3._default,0._default), (0._default,1._default), (1._default,2._default)/))
call evaluator_receive_momenta (eval)
call eval%evaluate ()
call interaction_write (int1, unit = u)
write (u, "(A)")
call interaction_write (int2, unit = u)
write (u, "(A)")
call eval%write (unit = u)
print *
call interaction_final (int1)
call interaction_final (int2)
call evaluator_final (eval)

print *
print *, "*** Evaluator for matrix square"
allocate (flv(4), col(4), qn(4))
call interaction_init (int1, 2, 0, 2, set_relations=.true.)
call flavor_init (flv, (/1, -1, 21, 21/), model)
call color_init (col(1), (/1/))
call color_init (col(2), (/-2/))
call color_init (col(3), (/2, -3/))
call color_init (col(4), (/3, -1/))
call quantum_numbers_init (qn, flv, col)
call interaction_add_state (int1, qn)
call color_init (col(3), (/3, -1/))
call color_init (col(4), (/2, -3/))
call quantum_numbers_init (qn, flv, col)
call interaction_add_state (int1, qn)
call color_init (col(3), (/2, -1/))
call color_init (col(4), .true.)
call quantum_numbers_init (qn, flv, col)
call interaction_add_state (int1, qn)
call interaction_freeze (int1)
! [qn_mask2 not set since default is false]
call evaluator_init_square (eval, int1, qn_mask2, nc=3)
call evaluator_init_square_nondiag (eval2, int1, qn_mask2)
qn_mask2 = new_quantum_numbers_mask (.false., .true., .true.)
call evaluator_init_square_diag (eval3, eval%int, qn_mask2)
call interaction_set_matrix_element &
      (int1, (/(2._default,0._default), (4._default,1._default), (-3._default,0._default)/))
call interaction_set_momenta (int1, p)
call interaction_write (int1, unit = u)
print *
call evaluator_receive_momenta (eval)
call eval%evaluate ()
call eval%write (unit = u)
print *
call evaluator_receive_momenta (eval2)
call eval2%evaluate ()
call eval2%write (unit = u)
print *
call evaluator_receive_momenta (eval3)
```

```
      call eval3%evaluate ()
      call eval3%write (unit = u)
      call interaction_final (int1)
      call evaluator_final (eval)
      call evaluator_final (eval2)
      call evaluator_final (eval3)
    end subroutine evaluator_1
```

⟨*Evaluators: execute tests*⟩+≡
```
    call test (evaluator_2, "evaluator_2", &
        "check evaluators (2)", &
        u, results)
```
⟨*Evaluators: tests*⟩+≡
```
    subroutine evaluator_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(interaction_t), target :: int
      integer :: h1, h2, h3, h4
      type(helicity_t), dimension(4) :: hel
      type(color_t), dimension(4) :: col
      type(flavor_t), dimension(4) :: flv
      type(quantum_numbers_t), dimension(4) :: qn
      type(vector4_t), dimension(4) :: p
      type(evaluator_t) :: eval
      integer :: i

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("SM"), &
          var_str ("SM.mdl"), os_data, model)

      write (u, "(A)") "*** Creating interaction for e+ e- -> W+ W-"
      write (u, "(A)")

      call flavor_init (flv, [11, -11, 24, -24], model)
      do i = 1, 4
         call color_init (col (i))
      end do
      call interaction_init (int, 2, 0, 2, set_relations=.true.)
      do h1 = -1, 1, 2
         call helicity_init (hel(1), h1)
         do h2 = -1, 1, 2
            call helicity_init (hel(2), h2)
            do h3 = -1, 1
               call helicity_init (hel(3), h3)
               do h4 = -1, 1
                  call helicity_init (hel(4), h4)
                  call quantum_numbers_init (qn, flv, col, hel)
                  call interaction_add_state (int, qn)
               end do
            end do
```

874

```
          end do
        end do
        call interaction_freeze (int)
        call interaction_set_matrix_element (int, &
           (/(cmplx (i, kind=default), i = 1, 36)/)))
        p(1) = vector4_moving (1000._default, 1000._default, 3)
        p(2) = vector4_moving (1000._default, -1000._default, 3)
        p(3) = vector4_moving (1000._default, &
           sqrt (1E6_default - 80._default**2), 3)
        p(4) = p(1) + p(2) - p(3)
        call interaction_set_momenta (int, p)
        write (u, "(A)") "*** Setting up evaluator"
        write (u, "(A)")

        call evaluator_init_identity (eval, int)
        write (u, "(A)") "*** Transferring momenta and evaluating"
        write (u, "(A)")

        call evaluator_receive_momenta (eval)
        call eval%evaluate ()
        write (u, "(A)")  "*****************************************************"
        write (u, "(A)")  "   Interaction dump"
        write (u, "(A)")  "*****************************************************"
        call interaction_write (int, unit = u)
        write (u, "(A)")
        write (u, "(A)")  "*****************************************************"
        write (u, "(A)")  "   Evaluator dump"
        write (u, "(A)")  "*****************************************************"
        call eval%write (unit = u)
        write (u, "(A)")
        write (u, "(A)")   "*** cleaning up"
        call interaction_final (int)
        call evaluator_final (eval)
      end subroutine evaluator_2


⟨Evaluators: execute tests⟩+≡
      call test (evaluator_3, "evaluator_3", &
          "check evaluators (3)", &
          u, results)

⟨Evaluators: tests⟩+≡
      subroutine evaluator_3 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_list_t) :: model_list
        type(model_t), pointer :: model
        type(interaction_t), target :: int
        integer :: h1, h2, h3, h4
        type(helicity_t), dimension(4) :: hel
        type(color_t), dimension(4) :: col
        type(flavor_t), dimension(4) :: flv1, flv2
        type(quantum_numbers_t), dimension(4) :: qn
        type(vector4_t), dimension(4) :: p
        type(evaluator_t) :: eval1, eval2, eval3
```

```
type(quantum_numbers_mask_t), dimension(4) :: qn_mask
integer :: i

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("SM"), &
     var_str ("SM.mdl"), os_data, model)

write (u, "(A)")  "*** Creating interaction for e+/mu+ e-/mu- -> W+ W-"
call flavor_init (flv1, (/11, -11, 24, -24/), model)
call flavor_init (flv2, (/13, -13, 24, -24/), model)
do i = 1, 4
   call color_init (col (i))
end do
call interaction_init (int, 2, 0, 2, set_relations=.true.)
do h1 = -1, 1, 2
   call helicity_init (hel(1), h1)
   do h2 = -1, 1, 2
      call helicity_init (hel(2), h2)
      do h3 = -1, 1
         call helicity_init (hel(3), h3)
         do h4 = -1, 1
            call helicity_init (hel(4), h4)
            call quantum_numbers_init (qn, flv1, col, hel)
            call interaction_add_state (int, qn)
            call quantum_numbers_init (qn, flv2, col, hel)
            call interaction_add_state (int, qn)
         end do
      end do
   end do
end do
call interaction_freeze (int)
call interaction_set_matrix_element (int, &
  (/(cmplx (1, kind=default), i = 1, 72)/))
p(1) = vector4_moving (1000._default, 1000._default, 3)
p(2) = vector4_moving (1000._default, -1000._default, 3)
p(3) = vector4_moving (1000._default, &
   sqrt (1E6_default - 80._default**2), 3)
p(4) = p(1) + p(2) - p(3)
call interaction_set_momenta (int, p)
write (u, "(A)")  "*** Setting up evaluators"
call quantum_numbers_mask_init (qn_mask, .false., .true., .true.)
call evaluator_init_qn_sum (eval1, int, qn_mask)
call quantum_numbers_mask_init (qn_mask, .true., .true., .true.)
call evaluator_init_qn_sum (eval2, int, qn_mask)
call quantum_numbers_mask_init (qn_mask, .false., .true., .false.)
call evaluator_init_qn_sum (eval3, int, qn_mask, &
  (/.false., .false., .false., .true./))
write (u, "(A)")  "*** Transferring momenta and evaluating"
call evaluator_receive_momenta (eval1)
call eval1%evaluate ()
call evaluator_receive_momenta (eval2)
call eval2%evaluate ()
call evaluator_receive_momenta (eval3)
```

```
    call eval3%evaluate ()
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Interaction dump"
    write (u, "(A)")  "*******************************************************"
    call interaction_write (int, unit = u)
    write (u, "(A)")
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Evaluator dump --- spin sum"
    write (u, "(A)")  "*******************************************************"
    call eval1%write (unit = u)
    call interaction_write (evaluator_get_int_ptr (eval1), unit = u)
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Evaluator dump --- spin / flavor sum"
    write (u, "(A)")  "*******************************************************"
    call eval2%write (unit = u)
    call interaction_write (evaluator_get_int_ptr (eval2), unit = u)
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Evaluator dump --- flavor sum, drop last W"
    write (u, "(A)")  "*******************************************************"
    call eval3%write (unit = u)
    call interaction_write (evaluator_get_int_ptr (eval3), unit = u)
    write (u, "(A)")
    write (u, "(A)")  "*** cleaning up"
    call interaction_final (int)
    call evaluator_final (eval1)
    call evaluator_final (eval2)
    call evaluator_final (eval3)
  end subroutine evaluator_3
```

Chapter 8

Particles

In this chapter, we deal with particles which have well-defined quantum numbers. While within interactions, all correlations are manifest, a particle array is derived by selecting a particular quantum number set. This involves tracing over all other particles, as far as polarization is concerned. Thus, a particle has definite flavor, color, and a single-particle density matrix for polarization.

8.1 Polarization

Particle polarization is determined by a particular quantum state which has just helicity information. For defining polarizations, we adopt the phase convention for a spin-1/2 particle that

$$\rho = \tfrac{1}{2}(1 + \vec{\alpha} \cdot \vec{\sigma}) \tag{8.1}$$

with the polarization axis $\vec{\alpha}$. Using this, we define

1. Trivial polarization: $\vec{\alpha} = 0$. [This is unpolarized, but distinct from the particular undefined polarization matrix which has the same meaning.]

2. Circular polarization: $\vec{\alpha}$ points in $\pm z$ direction.

3. Transversal polarization: $\vec{\alpha}$ points orthogonal to the z direction, with a phase ϕ that is 0 for the x axis, and $\pi/2 = 90°$ for the y axis. For antiparticles, the phase switches sign, corresponding to complex conjugation.

4. Axis polarization, where we explicitly give $\vec{\alpha}$.

For higher spin, we retain this definition, but apply it to the two components with maximum and minimum weight. For massless particles, this is sufficient. For massive particles, we add the possibilities:

5. Longitudinal polarization: Only the 0-component is set. This is possible only for bosons.

6. Diagonal polarization: Explicitly specify all components in the helicity basis.

Obviously, this does not exhaust the possible density matrices for higher spin, but it should cover all practical applications.

⟨polarizations.f90⟩≡
  ⟨*File header*⟩

  module polarizations

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants, only: imago !NODEP!
  ⟨*Use file utils*⟩
    use limits, only: FMT_12, FMT_19 !NODEP!
    use lorentz !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices

  ⟨*Standard module head*⟩

  ⟨*Polarizations: public*⟩

  ⟨*Polarizations: types*⟩

  ⟨*Polarizations: interfaces*⟩

  contains

  ⟨*Polarizations: procedures*⟩

  ⟨*Polarizations: tests*⟩

  end module polarizations

8.1.1 The polarization type

This is not an extension, but rather a restriction of the quantum state. Flavor and color are ignored, there is just a one-particle helicity density matrix.

⟨*Polarizations: public*⟩≡
    public :: polarization_t

⟨*Polarizations: types*⟩≡
    type :: polarization_t
       logical :: polarized = .false.
       integer :: spin_type = 0
       integer :: multiplicity = 0
       type(state_matrix_t) :: state
    end type polarization_t

8.1.2 Basic initializer and finalizer

We need the particle flavor for determining the allowed helicity values. The density matrix is not set, but prepared to be filled later. This is private.

⟨*Polarizations: procedures*⟩≡
```
elemental subroutine polarization_init (pol, flv)
  type(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  pol%spin_type = flavor_get_spin_type (flv)
  pol%multiplicity = flavor_get_multiplicity (flv)
  call state_matrix_init (pol%state, store_values = .true.)
end subroutine polarization_init
```

The finalizer has to be public. The quantum state contains memory allocated to pointers.

⟨*Polarizations: public*⟩+≡
```
public :: polarization_final
```

⟨*Polarizations: procedures*⟩+≡
```
elemental subroutine polarization_final (pol)
  type(polarization_t), intent(inout) :: pol
  call state_matrix_final (pol%state)
end subroutine polarization_final
```

8.1.3 I/O

⟨*Polarizations: public*⟩+≡
```
public :: polarization_write
```

⟨*Polarizations: procedures*⟩+≡
```
subroutine polarization_write (pol, unit)
  type(polarization_t), intent(in) :: pol
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A,I1,A,I1,A)")  &
       "Polarization: [spin_type = ", pol%spin_type, &
       ", mult = ", pol%multiplicity, "]"
  call state_matrix_write (pol%state, unit=unit)
end subroutine polarization_write
```

Defined assignment: deep copy

⟨*Polarizations: public*⟩+≡
```
public :: assignment(=)
```

⟨*Polarizations: interfaces*⟩≡
```
interface assignment(=)
   module procedure polarization_assign
end interface
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_assign (pol_out, pol_in)
      type(polarization_t), intent(out) :: pol_out
      type(polarization_t), intent(in) :: pol_in
      pol_out%polarized = pol_in%polarized
      pol_out%spin_type = pol_in%spin_type
      pol_out%multiplicity = pol_in%multiplicity
      pol_out%state = pol_in%state
    end subroutine polarization_assign
```

Binary I/O.
    Note: currently disabled; state matrix raw I/O questionable.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_write_raw
    public :: polarization_read_raw
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_write_raw (pol, u)
      type(polarization_t), intent(in) :: pol
      integer, intent(in) :: u
      write (u) pol%polarized
      write (u) pol%spin_type
      write (u) pol%multiplicity
      if (state_matrix_is_defined (pol%state)) then
         call msg_bug ("Write polarization (raw): nontrivial state matrix not supported yet")
      else
         call state_matrix_write_raw (pol%state, u)
      end if
    end subroutine polarization_write_raw

    subroutine polarization_read_raw (pol, u, iostat)
      type(polarization_t), intent(out) :: pol
      integer, intent(in) :: u
      integer, intent(out), optional :: iostat
      read (u, iostat=iostat) pol%polarized
      read (u, iostat=iostat) pol%spin_type
      read (u, iostat=iostat) pol%multiplicity
      call state_matrix_read_raw (pol%state, u, iostat=iostat)
    end subroutine polarization_read_raw
```

8.1.4 Accessing contents

Return true if the particle is polarized. This is the case if the first (and only) entry in the quantum state has undefined helicity.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_is_polarized
```

⟨*Polarizations: procedures*⟩+≡
```
    elemental function polarization_is_polarized (pol) result (polarized)
      logical :: polarized
      type(polarization_t), intent(in) :: pol
      polarized = pol%polarized
```

```
      end function polarization_is_polarized
```

Return true if the polarization is diagonal, i.e., all entries in the density matrix are diagonal.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_is_diagonal
```

⟨*Polarizations: interfaces*⟩+≡
```
    interface polarization_is_diagonal
       module procedure polarization_is_diagonal0
       module procedure polarization_is_diagonal1
    end interface
```

⟨*Polarizations: procedures*⟩+≡
```
    function polarization_is_diagonal0 (pol) result (diagonal)
      logical :: diagonal
      type(polarization_t), intent(in) :: pol
      type(state_iterator_t) :: it
      diagonal = .true.
      call state_iterator_init (it, pol%state)
      do while (state_iterator_is_valid (it))
         diagonal = all (quantum_numbers_are_diagonal &
              (state_iterator_get_quantum_numbers (it)))
         if (.not. diagonal) exit
         call state_iterator_advance (it)
      end do
    end function polarization_is_diagonal0

    function polarization_is_diagonal1 (pol) result (diagonal)
      type(polarization_t), dimension(:), intent(in) :: pol
      logical, dimension(size(pol)) :: diagonal
      integer :: i
      do i = 1, size (pol)
         diagonal(i) = polarization_is_diagonal0 (pol(i))
      end do
    end function polarization_is_diagonal1
```

8.1.5   Initialization from state matrix

Here, the state matrix is already known (but not necessarily normalized). The result will be either unpolarized, or a normalized spin density matrix.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_state_matrix
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_state_matrix (pol, state)
      type(polarization_t), intent(out) :: pol
      type(state_matrix_t), intent(in), target :: state
      type(state_iterator_t) :: it
      type(flavor_t) :: flv
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
```

```
   complex(default) :: value, t
   call state_iterator_init (it, state)
   flv = state_iterator_get_flavor (it, 1)
   hel = state_iterator_get_helicity (it, 1)
   if (helicity_is_defined (hel)) then
      call polarization_init (pol, flv)
      pol%polarized = .true.
      t = 0
      do while (state_iterator_is_valid (it))
         hel = state_iterator_get_helicity (it, 1)
         call quantum_numbers_init (qn(1), hel)
         value = state_iterator_get_matrix_element (it)
         call state_matrix_add_state (pol%state, qn, value=value)
         if (helicity_is_diagonal (hel))  t = t + value
         call state_iterator_advance (it)
      end do
      call state_matrix_freeze (pol%state)
      if (t /= 0)  call state_matrix_renormalize (pol%state, 1._default / t)
   else
      call polarization_init_unpolarized (pol, flv)
   end if
end subroutine polarization_init_state_matrix
```

8.1.6 Specific initializers

Unpolarized particle, no helicity labels in the density matrix. The value is
specified as $1/N$, where N is the multiplicity.

Exception: for left-handed or right-handed particles (neutrinos), polarization
is always circular with fraction unity.

⟨*Polarizations: public*⟩+≡
```
   public :: polarization_init_unpolarized
```

⟨*Polarizations: procedures*⟩+≡
```
   subroutine polarization_init_unpolarized (pol, flv)
     type(polarization_t), intent(inout) :: pol
     type(flavor_t), intent(in) :: flv
     type(quantum_numbers_t), dimension(1) :: qn
     complex(default) :: value
     if (flavor_is_left_handed (flv)) then
        call polarization_init_circular (pol, flv, -1._default)
     else if (flavor_is_right_handed (flv)) then
        call polarization_init_circular (pol, flv, 1._default)
     else
        call polarization_init (pol, flv)
        value = 1._default / flavor_get_multiplicity (flv)
        call state_matrix_add_state (pol%state, qn)
        call state_matrix_freeze (pol%state)
        call state_matrix_set_matrix_element (pol%state, value)
     end if
   end subroutine polarization_init_unpolarized
```

Unpolarized particle, but explicit density matrix with helicity states allocated according to given flavor. Note that fermions have even spin type, bosons odd. The spin density matrix entries are scaled by `fraction`. This is used for initializing other polarizations:

$$\rho(f) = \frac{|f|}{N}\mathbf{1}.$$

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_trivial
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_trivial (pol, flv, fraction)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), intent(in), optional :: fraction
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      integer :: h, hmax
      logical :: fermion
      complex(default) :: value
      call polarization_init (pol, flv)
      pol%polarized = .true.
      if (present (fraction)) then
         value = fraction / pol%multiplicity
      else
         value = 1._default / pol%multiplicity
      end if
      fermion = mod (pol%spin_type, 2) == 0
      hmax = pol%spin_type / 2
      select case (pol%multiplicity)
      case (1)
         if (flavor_is_left_handed (flv)) then
            call helicity_init (hel, -hmax)
         else if (flavor_is_right_handed (flv)) then
            call helicity_init (hel, hmax)
         else
            call helicity_init (hel, 0)
         end if
         call quantum_numbers_init (qn(1), hel)
         call state_matrix_add_state (pol%state, qn)
      case (2)
         do h = -hmax, hmax, 2*hmax
            call helicity_init (hel, h)
            call quantum_numbers_init (qn(1), hel)
            call state_matrix_add_state (pol%state, qn)
         end do
      case default
         do h = -hmax, hmax
            if (fermion .and. h == 0)  cycle
            call helicity_init (hel, h)
            call quantum_numbers_init (qn(1), hel)
            call state_matrix_add_state (pol%state, qn)
         end do
      end select
      call state_matrix_freeze (pol%state)
```

```
      call state_matrix_set_matrix_element (pol%state, value)
    end subroutine polarization_init_trivial
```

The following three modes are useful mainly for spin-1/2 particle and massless particles of any nonzero spin. Only the highest-weight components are filled.

Circular polarization: The density matrix of the two highest-weight states is

$$\rho(f) = \frac{1-|f|}{2}\mathbf{1} + |f| \times \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, & f > 0; \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, & f < 0, \end{cases}$$

If the polarization fraction $|f|$ is unity, we need only one entry in the density matrix.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_circular
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_circular (pol, flv, fraction)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), intent(in) :: fraction
      type(helicity_t), dimension(2) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      complex(default) :: value
      integer :: hmax
      call polarization_init (pol, flv)
      pol%polarized = .true.
      hmax = pol%spin_type / 2
      call helicity_init (hel(1), hmax)
      call helicity_init (hel(2),-hmax)
      if (abs (fraction) /= 1) then
         value = (1 + fraction) / 2
         call quantum_numbers_init (qn(1), hel(1))
         call state_matrix_add_state (pol%state, qn, value=value)
         value = (1 - fraction) / 2
         call quantum_numbers_init (qn(1), hel(2))
         call state_matrix_add_state (pol%state, qn, value=value)
      else
         value = abs (fraction)
         if (fraction > 0) then
            call quantum_numbers_init (qn(1), hel(1))
         else
            call quantum_numbers_init (qn(1), hel(2))
         end if
         call state_matrix_add_state (pol%state, qn, value=value)
      end if
      call state_matrix_freeze (pol%state)
    end subroutine polarization_init_circular
```

Transversal polarization is analogous to circular, but we get a density matrix

$$\rho(f, \phi) = \frac{1 - |f|}{2} \mathbf{1} + \frac{|f|}{2} \begin{pmatrix} 1 & e^{-i\phi} \\ e^{i\phi} & 1 \end{pmatrix}.$$

The phase is $\phi = 0$ for the x-axis, $\phi = 90°$ for the y axis as polarization vector. For an antiparticle, the phase switches sign, and for $f < 0$, the off-diagonal elements switch sign.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_transversal
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_transversal (pol, flv, phi, fraction)
      type(polarization_t), intent(inout) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), intent(in) :: phi, fraction
      call polarization_init_axis &
           (pol, flv, fraction * (/ cos (phi), sin (phi), 0._default/))
    end subroutine polarization_init_transversal
```

For axis polarization, we again set only the entries with maximum weight.

$$\rho(f, \phi) = \frac{1}{2} \begin{pmatrix} 1 + \alpha_3 & \alpha_1 - i\alpha_2 \\ \alpha_1 + i\alpha_2 & 1 - \alpha_3 \end{pmatrix}.$$

For an antiparticle, α_2 switches sign (complex conjugate).

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_axis
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_axis (pol, flv, alpha)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), dimension(3), intent(in) :: alpha
      type(quantum_numbers_t), dimension(1) :: qn
      type(helicity_t), dimension(2,2) :: hel
      complex(default), dimension(2,2) :: value
      integer :: hmax
      call polarization_init (pol, flv)
      pol%polarized = .true.
      hmax = pol%spin_type / 2
      call helicity_init (hel(1,1), hmax, hmax)
      call helicity_init (hel(1,2), hmax,-hmax)
      call helicity_init (hel(2,1),-hmax, hmax)
      call helicity_init (hel(2,2),-hmax,-hmax)
      value(1,1) = (1 + alpha(3)) / 2
      value(2,2) = (1 - alpha(3)) / 2
      if (flavor_is_antiparticle (flv)) then
         value(1,2) = (alpha(1) + imago * alpha(2)) / 2
      else
         value(1,2) = (alpha(1) - imago * alpha(2)) / 2
      end if
      value(2,1) = conjg (value(1,2))
      if (value(1,1) /= 0) then
```

```
      call quantum_numbers_init (qn(1), hel(1,1))
      call state_matrix_add_state (pol%state, qn, value=value(1,1))
   end if
   if (value(2,2) /= 0) then
      call quantum_numbers_init (qn(1), hel(2,2))
      call state_matrix_add_state (pol%state, qn, value=value(2,2))
   end if
   if (value(1,2) /= 0) then
      call quantum_numbers_init (qn(1), hel(1,2))
      call state_matrix_add_state (pol%state, qn, value=value(1,2))
      call quantum_numbers_init (qn(1), hel(2,1))
      call state_matrix_add_state (pol%state, qn, value=value(2,1))
   end if
   call state_matrix_freeze (pol%state)
 end subroutine polarization_init_axis
```

This version specifies the polarization axis in terms of r (polarization degree) and θ, ϕ (polar and azimuthal angles).

If one of the angles is a nonzero multiple of π, roundoff errors typically will result in tiny contributions to unwanted components. Therefore, include a catch for small numbers.

⟨*Polarizations: public*⟩+≡
```
   public :: polarization_init_angles
```

⟨*Polarizations: procedures*⟩+≡
```
   subroutine polarization_init_angles (pol, flv, r, theta, phi)
     type(polarization_t), intent(out) :: pol
     type(flavor_t), intent(in) :: flv
     real(default), intent(in) :: r, theta, phi
     real(default), dimension(3) :: alpha
     real(default), parameter :: eps = 10 * epsilon (1._default)
     alpha(1) = r * sin (theta) * cos (phi)
     alpha(2) = r * sin (theta) * sin (phi)
     alpha(3) = r * cos (theta)
     where (abs (alpha) < eps)  alpha = 0
     call polarization_init_axis (pol, flv, alpha)
   end subroutine polarization_init_angles
```

Longitudinal polarization is defined only for massive bosons. Only the zero component is filled. Otherwise, unpolarized.

⟨*Polarizations: public*⟩+≡
```
   public :: polarization_init_longitudinal
```

⟨*Polarizations: procedures*⟩+≡
```
   subroutine polarization_init_longitudinal (pol, flv, fraction)
     type(polarization_t), intent(out) :: pol
     type(flavor_t), intent(in) :: flv
     real(default), intent(in) :: fraction
     integer :: spin_type, multiplicity
     type(helicity_t) :: hel
     type(quantum_numbers_t), dimension(1) :: qn
     complex(default) :: value
     integer :: n_values
```

```
            value = abs (fraction)
            spin_type = flavor_get_spin_type (flv)
            multiplicity = flavor_get_multiplicity (flv)
            if (mod (spin_type, 2) == 1 .and. multiplicity > 2) then
               if (fraction /= 1) then
                  call polarization_init_trivial (pol, flv, 1 - fraction)
                  n_values = state_matrix_get_n_matrix_elements (pol%state)
                  call state_matrix_add_to_matrix_element &
                       (pol%state, n_values/2 + 1, value)
               else
                  call polarization_init (pol, flv)
                  pol%polarized = .true.
                  call helicity_init (hel, 0)
                  call quantum_numbers_init (qn(1), hel)
                  call state_matrix_add_state (pol%state, qn)
                  call state_matrix_freeze (pol%state)
                  call state_matrix_set_matrix_element (pol%state, value)
               end if
            else
               call polarization_init_unpolarized (pol, flv)
            end if
          end subroutine polarization_init_longitudinal
```

This is diagonal polarization: we specify all components explicitly. We use only
the positive components. The sum is normalized to unity. We assume that the
length of `alpha` is equal to the particle multiplicity.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_diagonal
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_diagonal (pol, flv, alpha)
      type(polarization_t), intent(inout) :: pol
      type(flavor_t), intent(in) :: flv
      real(default), dimension(:), intent(in) :: alpha
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      logical, dimension(size(alpha)) :: mask
      real(default) :: norm
      complex(default), dimension(:), allocatable :: value
      logical :: fermion
      integer :: h, hmax, i
      mask = alpha > 0
      norm = sum (alpha, mask);  if (norm == 0)  norm = 1
      allocate (value (count (mask)))
      value = pack (alpha / norm, mask)
      call polarization_init (pol, flv)
      pol%polarized = .true.
      fermion = mod (pol%spin_type, 2) == 0
      hmax = pol%spin_type / 2
      i = 0
      select case (pol%multiplicity)
      case (1)
         if (flavor_is_left_handed (flv)) then
            call helicity_init (hel, -hmax)
```

```
         else if (flavor_is_right_handed (flv)) then
            call helicity_init (hel, hmax)
         else
            call helicity_init (hel, 0)
         end if
         call quantum_numbers_init (qn(1), hel)
         call state_matrix_add_state (pol%state, qn)
      case (2)
         do h = -hmax, hmax, 2*hmax
            i = i + 1
            if (mask(i)) then
               call helicity_init (hel, h)
               call quantum_numbers_init (qn(1), hel)
               call state_matrix_add_state (pol%state, qn)
            end if
         end do
      case default
         do h = -hmax, hmax
            if (fermion .and. h == 0)  cycle
            i = i + 1
            if (mask(i)) then
               call helicity_init (hel, h)
               call quantum_numbers_init (qn(1), hel)
               call state_matrix_add_state (pol%state, qn)
            end if
         end do
      end select
      call state_matrix_freeze (pol%state)
      call state_matrix_set_matrix_element (pol%state, value)
   end subroutine polarization_init_diagonal
```

Generic polarization: we generate all possible density matrix entries, but the values are left zero.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_init_generic
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_init_generic (pol, flv)
      type(polarization_t), intent(out) :: pol
      type(flavor_t), intent(in) :: flv
      type(helicity_t) :: hel
      type(quantum_numbers_t), dimension(1) :: qn
      logical :: fermion
      integer :: hmax, h1, h2
      call polarization_init (pol, flv)
      pol%polarized = .true.
      fermion = mod (pol%spin_type, 2) == 0
      hmax = pol%spin_type / 2
      select case (pol%multiplicity)
      case (1)
         if (flavor_is_left_handed (flv)) then
            call helicity_init (hel, -hmax)
         else if (flavor_is_right_handed (flv)) then
            call helicity_init (hel, hmax)
```

```
                else
                   call helicity_init (hel, 0)
                end if
                call quantum_numbers_init (qn(1), hel)
                call state_matrix_add_state (pol%state, qn)
             case (2)
                do h1 = -hmax, hmax, 2*hmax
                   do h2 = -hmax, hmax, 2*hmax
                      call helicity_init (hel, h1, h2)
                      call quantum_numbers_init (qn(1), hel)
                      call state_matrix_add_state (pol%state, qn)
                   end do
                end do
             case default
                do h1 = -hmax, hmax
                   if (fermion .and. h1 == 0)  cycle
                   do h2 = -hmax, hmax
                      if (fermion .and. h2 == 0)  cycle
                      call helicity_init (hel, h1, h2)
                      call quantum_numbers_init (qn(1), hel)
                      call state_matrix_add_state (pol%state, qn)
                   end do
                end do
             end select
             call state_matrix_freeze (pol%state)
          end subroutine polarization_init_generic
```

8.1.7   Operations

Combine polarization states by computing the outer product of the state matrices.

⟨*Polarizations: public*⟩+≡
```
    public :: combine_polarization_states
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine combine_polarization_states (pol, state)
      type(polarization_t), dimension(:), intent(in), target :: pol
      type(state_matrix_t), intent(out) :: state
      call outer_multiply (pol%state, state)
    end subroutine combine_polarization_states
```

Transform a polarization density matrix into a polarization vector. This is possible without information loss only for spin-1/2 and for massless particles. To get a unique answer in all cases, we consider only the components with highest weight. Obviously, this loses the longitudinal component of a massive vector, for instance.

This is the inverse operation of `polarization_init_axis` above, where the polarization fraction is set to unity.

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_get_axis
```

⟨*Polarizations: procedures*⟩+≡
```
function polarization_get_axis (pol) result (alpha)
  real(default), dimension(3) :: alpha
  type(polarization_t), intent(in), target :: pol
  type(state_iterator_t) :: it
  complex(default), dimension(2,2) :: value
  type(helicity_t), dimension(2,2) :: hel
  type(helicity_t), dimension(1) :: hel1
  integer :: hmax, i, j
  if (pol%polarized) then
     hmax = pol%spin_type / 2
     call helicity_init (hel(1,1), hmax, hmax)
     call helicity_init (hel(1,2), hmax,-hmax)
     call helicity_init (hel(2,1),-hmax, hmax)
     call helicity_init (hel(2,2),-hmax,-hmax)
     value = 0
     call state_iterator_init (it, pol%state)
     do while (state_iterator_is_valid (it))
        hel1 = state_iterator_get_helicity (it)
        SCAN_HEL: do i = 1, 2
           do j = 1, 2
              if (hel1(1) == hel(i,j)) then
                 value(i,j) = state_iterator_get_matrix_element (it)
                 exit SCAN_HEL
              end if
           end do
        end do SCAN_HEL
        call state_iterator_advance (it)
     end do
     alpha(1) = value(1,2) + value(2,1)
     alpha(2) = imago * (value(1,2) - value(2,1))
     alpha(3) = value(1,1) - value(2,2)
  else
     alpha = 0
  end if
end function polarization_get_axis
```

This function returns polarization degree and polar and azimuthal angles (θ, ϕ) of the polarization axis.

⟨*Polarizations: public*⟩+≡
```
public :: polarization_to_angles
```

⟨*Polarizations: procedures*⟩+≡
```
subroutine polarization_to_angles (pol, r, theta, phi)
  type(polarization_t), intent(in) :: pol
  real(default), intent(out) :: r, theta, phi
  real(default), dimension(3) :: alpha
  real(default) :: r12
  if (pol%polarized) then
     alpha = polarization_get_axis (pol)
     r = sqrt (sum (alpha**2))
     if (any (alpha /= 0)) then
        r12 = sqrt (alpha(1)**2 + alpha(2)**2)
        theta = atan2 (r12, alpha(3))
```

```
            if (any (alpha(1:2) /= 0)) then
               phi = atan2 (alpha(2), alpha(1))
            else
               phi = 0
            end if
         else
            theta = 0
         end if
      else
         r = 0
         theta = 0
         phi = 0
      end if
   end subroutine polarization_to_angles
```

8.1.8 Sparse Matrix

We introduce a simpler implementation of a sparse matrix that should represent
polarization. It consists of an integer array that represents the index values,
and a complex array that represents the nonvanishing entries. The number of
nonvanishing entries must be known for initialization, but the entries are filled
one at a time.

This should eventually replace the more baroque polarization definition
above, but in the first stage we introduce it just as an intermediage storage
container.

Here is a base type without the special properties of a spin-density matrix.

⟨*Polarizations: public*⟩+≡
```
   public :: smatrix_t
```

⟨*Polarizations: types*⟩+≡
```
   type :: smatrix_t
     private
     integer :: dim = 0
     integer :: n_entry = 0
     integer, dimension(:,:), allocatable :: index
     complex(default), dimension(:), allocatable :: value
    contains
   ⟨Polarizations: smatrix: TBP⟩
   end type smatrix_t
```

Output.

⟨*Polarizations: smatrix: TBP*⟩≡
```
   procedure :: write => smatrix_write
```

⟨*Polarizations: procedures*⟩+≡
```
   subroutine smatrix_write (object, unit, indent)
     class(smatrix_t), intent(in) :: object
     integer, intent(in), optional :: unit, indent
     integer :: u, i, ind
     u = output_unit (unit)
     ind = 0;  if (present (indent))  ind = indent
     if (allocated (object%value)) then
```

```
            if (size (object%value) > 0) then
               do i = 1, object%n_entry
                  write (u, "(1x,A,'@(')", advance="no")  repeat ("  ", ind)
                  write (u, "(SP,9999(I2.1,':',1x))", advance="no") &
                       object%index(:,i)
                  write (u, "('(',"  // FMT_19 // ",',',"  // FMT_19 // &
                       ",')')')")  object%value(i)
               end do
            else
               write (u, "(1x,A)", advance="no")  repeat ("  ", ind)
               write (u, "(A)")  "[empty matrix]"
            end if
         else
            write (u, "(1x,A)", advance="no")  repeat ("  ", ind)
            write (u, "(A)")  "[undefined matrix]"
         end if
       end subroutine smatrix_write
```

Initialization: allocate arrays to the correct size. We specify both the dimension
of the matrix (if different from two, this is rather a generic tensor) and the
number of nonvanishing entries.

⟨*Polarizations: smatrix: TBP*⟩+≡
```
    procedure :: init => smatrix_init
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine smatrix_init (smatrix, dim, n_entry)
      class(smatrix_t), intent(out) :: smatrix
      integer, intent(in) :: dim
      integer, intent(in) :: n_entry
      smatrix%dim = dim
      smatrix%n_entry = n_entry
      allocate (smatrix%index (dim, n_entry))
      allocate (smatrix%value (n_entry))
    end subroutine smatrix_init
```

Fill: one entry at a time.

⟨*Polarizations: smatrix: TBP*⟩+≡
```
    procedure :: set_entry => smatrix_set_entry
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine smatrix_set_entry (smatrix, i, index, value)
      class(smatrix_t), intent(inout) :: smatrix
      integer, intent(in) :: i
      integer, dimension(:), intent(in) :: index
      complex(default), intent(in) :: value
      smatrix%index(:,i) = index
      smatrix%value(i) = value
    end subroutine smatrix_set_entry
```

8.1.9   Polarization Matrix

As an extension of the more generic `smatrix` type, we implement a proper
spin-density matrix. After the matrix has been filled, we can fix spin type and

multiplicity for a particle, check the matrix for consistency, and normalize it if
necessary.

⟨*Polarizations: public*⟩+≡
```
    public :: pmatrix_t
```

⟨*Polarizations: types*⟩+≡
```
    type, extends (smatrix_t) :: pmatrix_t
      private
      integer :: spin_type = 0
      integer :: multiplicity = 0
      logical :: massive = .true.
      integer :: chirality = 0
      real(default) :: degree = 1
      logical :: pure = .false.
    contains
    ⟨Polarizations: pmatrix: TBP⟩
    end type pmatrix_t
```

Output, including extra data. (The `indent` argument is ignored.)

⟨*Polarizations: pmatrix: TBP*⟩≡
```
    procedure :: write => pmatrix_write
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine pmatrix_write (object, unit, indent)
      class(pmatrix_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "Polarization: spin density matrix"
      write (u, "(3x,A,I0)")  "spin type     = ", object%spin_type
      write (u, "(3x,A,I0)")  "multiplicity  = ", object%multiplicity
      write (u, "(3x,A,L1)")  "massive       = ", object%massive
      write (u, "(3x,A,I0)")  "chirality     = ", object%chirality
      write (u, "(3x,A,F10.7)")  "pol.degree    =", object%degree
      write (u, "(3x,A,L1)")  "pure state    = ", object%pure
      call object%smatrix_t%write (u, 1)
    end subroutine pmatrix_write
```

This assignment is trivial, but must be coded explicitly.

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
    generic :: assignment(=) => pmatrix_assign_from_smatrix
    procedure, private :: pmatrix_assign_from_smatrix
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine pmatrix_assign_from_smatrix (pmatrix, smatrix)
      class(pmatrix_t), intent(out) :: pmatrix
      type(smatrix_t), intent(in) :: smatrix
      pmatrix%smatrix_t = smatrix
    end subroutine pmatrix_assign_from_smatrix
```

Declare spin, multiplicity, and polarization degree. Check whether all entries
fit, and whether this is a valid matrix.

The required properties are:

1. all entries apply to the given spin and mass type

2. the diagonal is real

3. only the upper of corresponding off-diagonal elements is specified, i.e., the row index is less than the column index

4. the trace is nonnegative and equal to the polarization degree (the remainder, proportional to the unit matrix, is understood to be present)

5. the trace of the matrix square is positive and less or equal to the trace of the matrix itself, which is the polarization degree. (If it is equal, we are dealing with a pure state).

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
procedure :: normalize => pmatrix_normalize
```

⟨*Polarizations: procedures*⟩+≡
```
subroutine pmatrix_normalize (pmatrix, flv, degree, tolerance)
  class(pmatrix_t), intent(inout) :: pmatrix
  type(flavor_t), intent(in) :: flv
  real(default), intent(in), optional :: degree
  real(default), intent(in), optional :: tolerance
  integer :: i, hmax
  logical :: fermion, ok
  real(default) :: trace, trace_sq
  real(default) :: tol
  tol = 0;  if (present (tolerance))  tol = tolerance
  pmatrix%spin_type = flavor_get_spin_type (flv)
  pmatrix%massive = flavor_get_mass (flv) /= 0
  if (.not. pmatrix%massive) then
     if (flavor_is_left_handed (flv)) then
        pmatrix%chirality = -1
     else if (flavor_is_right_handed (flv)) then
        pmatrix%chirality = +1
     end if
  end if
  if (pmatrix%spin_type == SCALAR) then
     pmatrix%multiplicity = 1
  else if (pmatrix%massive) then
     pmatrix%multiplicity = pmatrix%spin_type
  else if (pmatrix%chirality == 0) then
     pmatrix%multiplicity = 2
  else
     pmatrix%multiplicity = 1
  end if
  if (present (degree)) then
     if (degree < 0 .or. degree > 1) &
          call msg_error ("polarization degree must be between 0 and 1")
     pmatrix%degree = degree
  end if
  if (size (pmatrix%index, 1) /= 2)  call error ("wrong array rank")
  fermion = mod (pmatrix%spin_type, 2) == 0
  hmax = pmatrix%spin_type / 2
  if (pmatrix%n_entry > 0) then
     if (fermion) then
```

```fortran
      if (pmatrix%massive) then
         ok = all (pmatrix%index /= 0) &
              .and. all (abs (pmatrix%index) <= hmax)
      else if (pmatrix%chirality == -1) then
         ok = all (pmatrix%index == -hmax)
      else if (pmatrix%chirality == +1) then
         ok = all (pmatrix%index == +hmax)
      else
         ok = all (abs (pmatrix%index) == hmax)
      end if
   else
      if (pmatrix%massive) then
         ok = all (abs (pmatrix%index) <= hmax)
      else
         ok = all (abs (pmatrix%index) == hmax)
      end if
   end if
   if (.not. ok)  call error ("illegal index value")
else
   pmatrix%degree = 0
   pmatrix%pure = pmatrix%multiplicity == 1
   return
end if
trace = 0
do i = 1, pmatrix%n_entry
   associate (index => pmatrix%index(:,i), value => pmatrix%value(i))
     if (index(1) == index(2)) then
        if (abs (aimag (value)) > tol)  call error ("diagonal must be real")
        value = real (value, kind=default)
        trace = trace + value
     else if (any (pmatrix%index(1,:) == index(2) &
          .and.    pmatrix%index(2,:) == index(1))) then
        call error ("redundant off-diagonal entry")
     else if (index(2) < index (1)) then
        index = index([2,1])
        value = conjg (value)
     end if
   end associate
end do
if (abs (trace) <= tol)  call error ("trace must not vanish")
trace = real (trace, kind=default)
pmatrix%value = pmatrix%value / trace * pmatrix%degree
trace_sq = (1 - pmatrix%degree ** 2) / pmatrix%multiplicity
do i = 1, pmatrix%n_entry
   associate (index => pmatrix%index(:,i), value => pmatrix%value(i))
     if (index(1) == index(2)) then
        trace_sq = trace_sq + abs (value) ** 2
     else
        trace_sq = trace_sq + 2 * abs (value) ** 2
     end if
   end associate
end do
if (pmatrix%multiplicity == 1) then
   pmatrix%pure = .true.
```

```
         else if (abs (trace_sq - 1) <= tol) then
            pmatrix%pure = .true.
         else if (trace_sq - 1 > tol .or. trace_sq < -tol) then
            print *, "Trace of matrix square = ", trace_sq
            call error ("not permissible as density matrix")
         end if
      contains
        subroutine error (msg)
          character(*), intent(in) :: msg
          call pmatrix%write ()
          call msg_fatal ("Spin density matrix: " // msg)
        end subroutine error
      end subroutine pmatrix_normalize
```

A polarized matrix is defined as one with a positive polarization degree, even if
the actual matrix is trivial.

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
      procedure :: is_polarized => pmatrix_is_polarized
```

⟨*Polarizations: procedures*⟩+≡
```
      elemental function pmatrix_is_polarized (pmatrix) result (flag)
        class(pmatrix_t), intent(in) :: pmatrix
        logical :: flag
        flag = pmatrix%degree > 0
      end function pmatrix_is_polarized
```

Check if there are only diagonal entries.

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
      procedure :: is_diagonal => pmatrix_is_diagonal
```

⟨*Polarizations: procedures*⟩+≡
```
      elemental function pmatrix_is_diagonal (pmatrix) result (flag)
        class(pmatrix_t), intent(in) :: pmatrix
        logical :: flag
        flag = all (pmatrix%index(1,:) == pmatrix%index(2,:))
      end function pmatrix_is_diagonal
```

8.1.10 Data Translation

Create a `polarization_t` object from the contents of a normalized `pmatrix_t`
object.

Note for off diagonal entries: the convention for the arguments of `helicity_init`
is `(h2,h1)`, so the helicity of the conjugate state is inserted first.

⟨*Polarizations: public*⟩+≡
```
      public :: polarization_init_pmatrix
```

⟨*Polarizations: procedures*⟩+≡
```
      subroutine polarization_init_pmatrix (pol, pmatrix)
        type(polarization_t), intent(out) :: pol
        type(pmatrix_t), intent(in) :: pmatrix
        type(quantum_numbers_t), dimension(1) :: qn
        type(helicity_t) :: hel
```

897

```fortran
integer :: i, h, h1, h2, hmin, hmax, dh
logical :: fermion
complex(default) :: value
pol%polarized = .true.
pol%spin_type = pmatrix%spin_type
pol%multiplicity = pmatrix%multiplicity
call state_matrix_init (pol%state, store_values = .true.)
fermion = mod (pol%spin_type, 2) == 0
h = pol%spin_type / 2
select case (pmatrix%chirality)
case (-1)
   hmin = -h
   hmax = -h
case (0)
   hmin = -h
   hmax = h
case (1)
   hmin = h
   hmax = h
end select
if (pol%multiplicity == 1) then
   dh = 1
else if (pol%multiplicity == 2) then
   dh = hmax - hmin
else
   dh = 1
end if
if (pmatrix%degree < 1) then
   value = (1 - pmatrix%degree) / pol%multiplicity
   do h = hmin, hmax, dh
      if (h == 0 .and. fermion)  cycle
      call helicity_init (hel, h)
      call quantum_numbers_init (qn(1), hel)
      call state_matrix_add_state (pol%state, qn, value = value)
   end do
end if
do i = 1, pmatrix%n_entry
   associate (index => pmatrix%index(:,i), value => pmatrix%value(i))
     h1 = index(1)
     h2 = index(2)
     if (h1 == h2) then
        call helicity_init (hel, h1)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn, value = value, &
             sum_values = .true.)
     else
        call helicity_init (hel, h2, h1)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn, value = value)
        call helicity_init (hel, h1, h2)
        call quantum_numbers_init (qn(1), hel)
        call state_matrix_add_state (pol%state, qn, value = conjg (value))
     end if
   end associate
```

```
        end do
    call state_matrix_freeze (pol%state)
  end subroutine polarization_init_pmatrix
```

8.1.11 Unit tests

⟨*Polarizations: public*⟩+≡
```
    public :: polarization_test
```

⟨*Polarizations: procedures*⟩+≡
```
    subroutine polarization_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Polarizations: execute tests⟩
    end subroutine polarization_test
```

Polarization type

Checking the setup for polarization.

⟨*Polarizations: execute tests*⟩≡
```
    call test (polarization_1, "polarization_1", &
        "check polarization setup", &
        u, results)
```

⟨*Polarizations: tests*⟩≡
```
    subroutine polarization_1 (u)
      use os_interface
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(polarization_t) :: pol
      type(flavor_t) :: flv
      real(default), dimension(3) :: alpha
      real(default) :: r, theta, phi

      write (u, "(A)")  "* Test output: polarization_1"
      write (u, "(A)")  "*   Purpose: test polarization setup"
      write (u, "(A)")

      write (u, "(A)")  "* Reading model file"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
          (var_str("SM"), var_str("SM.mdl"), os_data, model)

      write (u, "(A)") "* Unpolarized fermion"
      write (u, "(A)")
```

```
call flavor_init (flv, 1, model)
call polarization_init_unpolarized (pol, flv)
call polarization_write (pol, u)
write (u, "(A,L1)") "   diagonal =", polarization_is_diagonal (pol)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Unpolarized fermion"
write (u, "(A)")

call polarization_init_circular (pol, flv, 0._default)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Transversally polarized fermion, phi=0"
write (u, "(A)")

call polarization_init_transversal (pol, flv, 0._default, 1._default)
call polarization_write (pol, u)
write (u, "(A,L1)") "   diagonal =", polarization_is_diagonal (pol)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Transversally polarized fermion, phi=0.9, frac=0.8"
write (u, "(A)")

call polarization_init_transversal (pol, flv, 0.9_default, 0.8_default)
call polarization_write (pol, u)
write (u, "(A,L1)") "   diagonal =", polarization_is_diagonal (pol)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)") "* All polarization directions of a fermion"
write (u, "(A)")

call polarization_init_generic (pol, flv)
call polarization_write (pol, u)
call polarization_final (pol)
call flavor_init (flv, 21, model)

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized gluon, frac=0.3"
write (u, "(A)")

call polarization_init_circular (pol, flv, 0.3_default)
call polarization_write (pol, u)
call polarization_final (pol)
call flavor_init (flv, 23, model)

write (u, "(A)")
write (u, "(A)") "* Circularly polarized massive vector, frac=-0.7"
write (u, "(A)")
```

```
call polarization_init_circular (pol, flv,  -0.7_default)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized massive vector"
write (u, "(A)")

call polarization_init_circular (pol, flv, 1._default)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector, frac=0.4"
write (u, "(A)")

call polarization_init_longitudinal (pol, flv, 0.4_default)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector"
write (u, "(A)")

call polarization_init_longitudinal (pol, flv, 1._default)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Diagonally polarized massive vector"
write (u, "(A)")

call polarization_init_diagonal &
     (pol, flv, [0._default, 1._default, 2._default])
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* All polarization directions of a massive vector"
write (u, "(A)")

call polarization_init_generic (pol, flv)
call polarization_write (pol, u)
call polarization_final (pol)
call flavor_init (flv, 21, model)

write (u, "(A)")
write (u, "(A)")  "* Axis polarization (0.2, 0.4, 0.6)"
write (u, "(A)")

alpha = [0.2_default, 0.4_default, 0.6_default]
call polarization_init_axis (pol, flv, alpha)
call polarization_write (pol, u)
```

901

```
    write (u, "(A)")  "   Recovered axis:"
    alpha = polarization_get_axis (pol)
    write (u, "(A)")  "   Angle polarization (0.5, 0.6, -1)"
    r = 0.5_default
    theta = 0.6_default
    phi = -1._default
    call polarization_init_angles (pol, flv, r, theta, phi)
    call polarization_write (pol, u)
    write (u, "(A)")  "   Recovered parameters (r, theta, phi):"
    call polarization_to_angles (pol, r, theta, phi)
    write (u, "(A,3(1x," // FMT_12 // "))")  "      ", r, theta, phi
    call polarization_final (pol)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: polarization_1"

  end subroutine polarization_1
```

Sparse-Matrix type

Use a sparse density matrix universally as the input for setting up polarization.

⟨*Polarizations: execute tests*⟩+≡
```
    call test (polarization_2, "polarization_2", &
         "matrix polarization setup", &
         u, results)
```

⟨*Polarizations: tests*⟩+≡
```
    subroutine polarization_2 (u)
      use os_interface
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(polarization_t) :: pol
      real(default), dimension(3) :: alpha
      type(pmatrix_t) :: pmatrix
      real(default), parameter :: tolerance = 1e-8_default

      write (u, "(A)")  "* Test output: polarization_2"
      write (u, "(A)")  "*   Purpose: matrix polarization setup"
      write (u, "(A)")

      write (u, "(A)")  "* Reading model file"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
           (var_str("SM"), var_str("SM.mdl"), os_data, model)

      write (u, "(A)") "* Unpolarized fermion"
      write (u, "(A)")
```

```
call flavor_init (flv, 1, model)
call pmatrix%init (2, 0)
call pmatrix%normalize (flv, 0._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Unpolarized fermion"
write (u, "(A)")

call pmatrix%init (2, 0)
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Transversally polarized fermion, phi=0"
write (u, "(A)")

call pmatrix%init (2, 3)
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%set_entry (2, [+1,+1], (1._default, 0._default))
call pmatrix%set_entry (3, [-1,+1], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Transversally polarized fermion, phi=0.9, frac=0.8"
write (u, "(A)")

call pmatrix%init (2, 3)
```

```
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%set_entry (2, [+1,+1], (1._default, 0._default))
call pmatrix%set_entry (3, [-1,+1], exp ((0._default, -0.9_default)))
call pmatrix%normalize (flv, 0.8_default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Left-handed massive fermion, frac=1"
write (u, "(A)")

call flavor_init (flv, 11, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Left-handed massive fermion, frac=0.8"
write (u, "(A)")

call flavor_init (flv, 11, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.8_default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Left-handed massless fermion"
write (u, "(A)")

call flavor_init (flv, 12, model)
call pmatrix%init (2, 0)
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
```

904

```
call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Right-handed massless fermion, frac=0.5"
write (u, "(A)")

call flavor_init (flv, -12, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.5_default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized gluon, frac=0.3"
write (u, "(A)")

call flavor_init (flv, 21, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.3_default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)") "* Circularly polarized massive vector, frac=0.7"
write (u, "(A)")

call flavor_init (flv, 23, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.7_default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized massive vector"
write (u, "(A)")

call flavor_init (flv, 23, model)
```

```
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector, frac=0.4"
write (u, "(A)")

call flavor_init (flv, 23, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [0,0], (1._default, 0._default))
call pmatrix%normalize (flv, 0.4_default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector"
write (u, "(A)")

call flavor_init (flv, 23, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [0,0], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Axis polarization (0.2, 0.4, 0.6)"
write (u, "(A)")

call flavor_init (flv, 11, model)
alpha = [0.2_default, 0.4_default, 0.6_default]
alpha = alpha / sqrt (sum (alpha**2))
call pmatrix%init (2, 3)
```

```
call pmatrix%set_entry (1, [-1,-1], cmplx (1 - alpha(3), kind=default))
call pmatrix%set_entry (2, [1,-1], &
     cmplx (alpha(1), -alpha(2), kind=default))
call pmatrix%set_entry (3, [1,1], cmplx (1 + alpha(3), kind=default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)

call polarization_init_pmatrix (pol, pmatrix)
call polarization_write (pol, u)
call polarization_final (pol)

write (u, "(A)")
write (u, "(A)")  "* Test output end: polarization_2"

end subroutine polarization_2
```

8.2   Particles

This module defines the `particle_t` object type, and the methods and operations that deal with it.

⟨particles.f90⟩≡
  ⟨*File header*⟩

```
module particles
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
  use limits, only: FMT_16, FMT_17, FMT_19 !NODEP!
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use unit_tests
  use subevents
  use expressions
  use models
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
  use interactions
  use evaluators
  use polarizations
  use hepmc_interface
```

  ⟨*Standard module head*⟩

  ⟨*Particles: public*⟩

  ⟨*Particles: parameters*⟩

907

⟨*Particles: types*⟩

⟨*Particles: interfaces*⟩

```
contains
```

⟨*Particles: procedures*⟩

⟨*Particles: tests*⟩

```
end module particles
```

8.2.1 The particle type

Particle status codes

The overall status codes (incoming/outgoing etc.) are inherited from the module `subevents`.

Polarization status:

⟨*Particles: parameters*⟩≡
```
integer, parameter, public :: PRT_UNPOLARIZED = 0
integer, parameter, public :: PRT_DEFINITE_HELICITY = 1
integer, parameter, public :: PRT_GENERIC_POLARIZATION = 2
```

Definition

The quantum numbers are flavor (from which invariant particle properties can be derived), color, and polarization. The particle may be unpolarized. In this case, `hel` and `pol` are unspecified. If it has a definite helicity, the `hel` component is defined. If it has a generic polarization, the `pol` component is defined. For each particle we store the four-momentum and the invariant mass squared, i.e., the squared norm of the four-momentum. There is also an optional list of parent and child particles, for bookkeeping in physical events.

⟨*Particles: public*⟩≡
```
public :: particle_t
```

⟨*Particles: types*⟩≡
```
type :: particle_t
   private
   integer :: status = PRT_UNDEFINED
   integer :: polarization = PRT_UNPOLARIZED
   type(flavor_t) :: flv
   type(color_t) :: col
   type(helicity_t) :: hel
   type(polarization_t) :: pol
   type(vector4_t) :: p = vector4_null
   real(default) :: p2 = 0
   integer, dimension(:), allocatable :: parent
   integer, dimension(:), allocatable :: child
 end type particle_t
```

Particle initializers:

⟨*Particles: interfaces*⟩≡
```
interface particle_init
   module procedure particle_init_particle
   module procedure particle_init_state
   module procedure particle_init_hepmc
end interface
```

Copy a particle. (Deep copy) The excludes the parent-child relations.

⟨*Particles: procedures*⟩≡
```
subroutine particle_init_particle (prt_out, prt_in)
  type(particle_t), intent(out) :: prt_out
  type(particle_t), intent(in) :: prt_in
  prt_out%status = prt_in%status
  prt_out%polarization = prt_in%polarization
  prt_out%flv = prt_in%flv
  prt_out%col = prt_in%col
  prt_out%hel = prt_in%hel
  prt_out%pol = prt_in%pol
  prt_out%p = prt_in%p
  prt_out%p2 = prt_in%p2
end subroutine particle_init_particle
```

Initialize a particle using a single-particle state matrix which determines flavor, color, and polarization. The state matrix must have unique flavor and color. The factorization mode determines whether the particle is unpolarized, has definite helicity, or generic polarization. This mode is translated into the polarization status.

⟨*Particles: procedures*⟩+≡
```
subroutine particle_init_state (prt, state, status, mode)
  type(particle_t), intent(out) :: prt
  type(state_matrix_t), intent(in) :: state
  integer, intent(in) :: status, mode
  type(state_iterator_t) :: it
  prt%status = status
  call state_iterator_init (it, state)
  prt%flv = state_iterator_get_flavor (it, 1)
  if (flavor_is_beam_remnant (prt%flv))  prt%status = PRT_BEAM_REMNANT
  prt%col = state_iterator_get_color (it, 1)
  select case (mode)
  case (FM_SELECT_HELICITY)
     prt%hel = state_iterator_get_helicity (it, 1)
     prt%polarization = PRT_DEFINITE_HELICITY
  case (FM_FACTOR_HELICITY)
     call polarization_init_state_matrix (prt%pol, state)
     prt%polarization = PRT_GENERIC_POLARIZATION
  end select
end subroutine particle_init_state
```

Initialize a particle from a HepMC particle object. The model is necessary for making a fully qualified flavor component. We have the additional flag

polarized which tells whether the polarization information should be interpreted or ignored, and the lookup array of barcodes. Note that the lookup array is searched linearly, a possible bottleneck for large particle arrays. If necessary, the barcode array could be replaced by a hash table.

⟨*Particles: procedures*⟩+≡

```
subroutine particle_init_hepmc (prt, hprt, model, polarization, barcode)
  type(particle_t), intent(out) :: prt
  type(hepmc_particle_t), intent(in) :: hprt
  type(model_t), intent(in), target :: model
  integer, intent(in) :: polarization
  integer, dimension(:), intent(in) :: barcode
  type(hepmc_polarization_t) :: hpol
  integer :: n_parents, n_children
  integer, dimension(:), allocatable :: parent_barcode, child_barcode
  integer :: i
  select case (hepmc_particle_get_status (hprt))
  case (1);  prt%status = PRT_OUTGOING
  case (2);  prt%status = PRT_RESONANT
  case (3);  prt%status = PRT_VIRTUAL
  end select
  call flavor_init (prt%flv, hepmc_particle_get_pdg (hprt), model)
  if (flavor_is_beam_remnant (prt%flv))  prt%status = PRT_BEAM_REMNANT
  call color_init (prt%col, hepmc_particle_get_color (hprt))
  prt%polarization = polarization
  select case (polarization)
  case (PRT_DEFINITE_HELICITY)
     hpol = hepmc_particle_get_polarization (hprt)
     call hepmc_polarization_to_hel (hpol, prt%flv, prt%hel)
     call hepmc_polarization_final (hpol)
  case (PRT_GENERIC_POLARIZATION)
     hpol = hepmc_particle_get_polarization (hprt)
     call hepmc_polarization_to_pol (hpol, prt%flv, prt%pol)
     call hepmc_polarization_final (hpol)
  end select
  prt%p = hepmc_particle_get_momentum (hprt)
  prt%p2 = hepmc_particle_get_mass_squared (hprt)
  n_parents  = hepmc_particle_get_n_parents  (hprt)
  n_children = hepmc_particle_get_n_children (hprt)
  allocate (parent_barcode (n_parents),  prt%parent (n_parents))
  allocate (child_barcode  (n_children), prt%child  (n_children))
  parent_barcode = hepmc_particle_get_parent_barcodes (hprt)
  child_barcode  = hepmc_particle_get_child_barcodes  (hprt)
  do i = 1, size (barcode)
     where (parent_barcode == barcode(i))  prt%parent = i
     where (child_barcode  == barcode(i))  prt%child  = i
  end do
  if (prt%status == PRT_VIRTUAL .and. n_parents == 0) &
       prt%status = PRT_INCOMING
end subroutine particle_init_hepmc
```

Finalizer. The polarization component has pointers allocated.

⟨*Particles: procedures*⟩+≡

```
elemental subroutine particle_final (prt)
```

```
      type(particle_t), intent(inout) :: prt
      call polarization_final (prt%pol)
    end subroutine particle_final
```

I/O

⟨*Particles: public*⟩+≡
```
    public :: particle_write
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_write (prt, unit, testflag)
      type(particle_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      logical :: pacified
      integer :: u
      real(default) :: pp2
      character(len=7) :: fmt
      pacified = .false.
      if (present (testflag))  pacified = testflag
      call pac_fmt (fmt, FMT_19, FMT_16, testflag)
      u = output_unit (unit); if (u < 0)  return
      select case (prt%status)
      case (PRT_UNDEFINED);    write (u, "(1x, A)", advance="no")  "[-]"
      case (PRT_BEAM);         write (u, "(1x, A)", advance="no")  "[b]"
      case (PRT_INCOMING);     write (u, "(1x, A)", advance="no")  "[i]"
      case (PRT_OUTGOING);     write (u, "(1x, A)", advance="no")  "[o]"
      case (PRT_VIRTUAL);      write (u, "(1x, A)", advance="no")  "[v]"
      case (PRT_RESONANT);     write (u, "(1x, A)", advance="no")  "[r]"
      case (PRT_BEAM_REMNANT); write (u, "(1x, A)", advance="no")  "[x]"
      end select
      write (u, "(1x)", advance="no")
      call flavor_write (prt%flv, unit)
      call color_write (prt%col, unit)
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call helicity_write (prt%hel, unit)
         write (u, *)
      case (PRT_GENERIC_POLARIZATION)
         write (u, *)
         call polarization_write (prt%pol, unit)
      case default
         write (u, *)
      end select
      call vector4_write (prt%p, unit, testflag = testflag)
      pp2 = prt%p2
      if (pacified)  call pacify (pp2, tolerance = 1E-10_default)
      write (u, "(1x,A,1x," // fmt // ")")  "T = ", pp2
      if (allocated (prt%parent)) then
         if (size (prt%parent) /= 0) then
            write (u, "(1x,A,40(1x,I0))")  "Parents: ", prt%parent
         end if
      end if
      if (allocated (prt%child)) then
```

911

```
        if (size (prt%child) /= 0) then
            write (u, "(1x,A,40(1x,I0))")  "Children:", prt%child
        end if
      end if
    end subroutine particle_write
```

Binary I/O:

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_write_raw (prt, u)
      type(particle_t), intent(in) :: prt
      integer, intent(in) :: u
      write (u) prt%status, prt%polarization
      call flavor_write_raw (prt%flv, u)
      call color_write_raw (prt%col, u)
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call helicity_write_raw (prt%hel, u)
      case (PRT_GENERIC_POLARIZATION)
         call polarization_write_raw (prt%pol, u)
      end select
      call vector4_write_raw (prt%p, u)
      write (u) prt%p2
      write (u) allocated (prt%parent)
      if (allocated (prt%parent)) then
         write (u) size (prt%parent)
         write (u) prt%parent
      end if
      write (u) allocated (prt%child)
      if (allocated (prt%child)) then
         write (u) size (prt%child)
         write (u) prt%child
      end if
    end subroutine particle_write_raw

    subroutine particle_read_raw (prt, u, iostat)
      type(particle_t), intent(out) :: prt
      integer, intent(in) :: u
      integer, intent(out) :: iostat
      logical :: allocated_parent, allocated_child
      integer :: size_parent, size_child
      read (u, iostat=iostat) prt%status, prt%polarization
      call flavor_read_raw (prt%flv, u, iostat=iostat)
      call color_read_raw (prt%col, u, iostat=iostat)
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call helicity_read_raw (prt%hel, u, iostat=iostat)
      case (PRT_GENERIC_POLARIZATION)
         call polarization_read_raw (prt%pol, u, iostat=iostat)
      end select
      call vector4_read_raw (prt%p, u, iostat=iostat)
      read (u, iostat=iostat) prt%p2
      read (u, iostat=iostat) allocated_parent
      if (allocated_parent) then
```

```
            read (u, iostat=iostat) size_parent
            allocate (prt%parent (size_parent))
            read (u, iostat=iostat) prt%parent
         end if
         read (u, iostat=iostat) allocated_child
         if (allocated_child) then
            read (u, iostat=iostat) size_child
            allocate (prt%child (size_child))
            read (u, iostat=iostat) prt%child
         end if
    end subroutine particle_read_raw
```

Setting contents

Reset the status code. Where applicable, set p^2 assuming that the particle is on-shell.

⟨*Particles: public*⟩+≡
```
    public :: particle_reset_status
```

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_reset_status (prt, status)
      type(particle_t), intent(inout) :: prt
      integer, intent(in) :: status
      prt%status = status
      select case (status)
      case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
         prt%p2 = flavor_get_mass (prt%flv) ** 2
      end select
    end subroutine particle_reset_status
```

The color can be given explicitly.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_color
```

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_set_color (prt, col)
      type(particle_t), intent(inout) :: prt
      type(color_t), intent(in) :: col
      prt%col = col
    end subroutine particle_set_color
```

The flavor can be given explicitly.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_flavor
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_flavor (prt, flv)
      type(particle_t), intent(inout) :: prt
      type(flavor_t), intent(in) :: flv
      prt%flv = flv
    end subroutine particle_set_flavor
```

Manually set the model for the particle flavor. This is required, e.g., if the particle has been read from file.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_model
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_model (prt, model)
      type(particle_t), intent(inout) :: prt
      type(model_t), intent(in), target :: model
      call flavor_set_model (prt%flv, model)
    end subroutine particle_set_model
```

The momentum is set independent of the quantum numbers.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_momentum
```

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_set_momentum (prt, p, p2, on_shell)
      type(particle_t), intent(inout) :: prt
      type(vector4_t), intent(in) :: p
      real(default), intent(in), optional :: p2
      logical, intent(in), optional :: on_shell
      prt%p = p
      if (present (on_shell)) then
         if (on_shell) then
            if (flavor_is_associated (prt%flv)) then
               prt%p2 = flavor_get_mass (prt%flv) ** 2
               return
            end if
         end if
      end if
      if (present (p2)) then
         prt%p2 = p2
      else
         prt%p2 = p ** 2
      end if
    end subroutine particle_set_momentum
```

Set resonance information. This should be done after momentum assignment, because we need to know wheter the particle is spacelike or timelike. The resonance flag is defined only for virtual particles.

⟨*Particles: procedures*⟩+≡
```
    elemental subroutine particle_set_resonance_flag (prt, resonant)
      type(particle_t), intent(inout) :: prt
      logical, intent(in) :: resonant
      select case (prt%status)
      case (PRT_VIRTUAL)
         if (resonant)  prt%status = PRT_RESONANT
      end select
    end subroutine particle_set_resonance_flag
```

Set children and parents information.

⟨Particles: public⟩+≡
```
    public :: particle_set_children
    public :: particle_set_parents
```

⟨Particles: procedures⟩+≡
```
    subroutine particle_set_children (prt, idx)
      type(particle_t), intent(inout) :: prt
      integer, dimension(:), intent(in) :: idx
      if (allocated (prt%child))  deallocate (prt%child)
      allocate (prt%child (count (idx /= 0)))
      prt%child = pack (idx, idx /= 0)
    end subroutine particle_set_children

    subroutine particle_set_parents (prt, idx)
      type(particle_t), intent(inout) :: prt
      integer, dimension(:), intent(in) :: idx
      if (allocated (prt%parent))  deallocate (prt%parent)
      allocate (prt%parent (count (idx /= 0)))
      prt%parent = pack (idx, idx /= 0)
    end subroutine particle_set_parents
```

Accessing contents

The status code.

⟨Particles: public⟩+≡
```
    public :: particle_get_status
```

⟨Particles: procedures⟩+≡
```
    elemental function particle_get_status (prt) result (status)
      integer :: status
      type(particle_t), intent(in) :: prt
      status = prt%status
    end function particle_get_status
```

Return true if the status is either `INCOMING`, `OUTGOING` or `RESONANT`. `BEAM` is kept, if `keep_beams` is set true.

⟨Particles: procedures⟩+≡
```
    elemental function particle_is_real (prt, keep_beams) result (flag)
      logical :: flag, kb
      type(particle_t), intent(in) :: prt
      logical, intent(in), optional :: keep_beams
      kb = .false.
      if (present (keep_beams)) kb = keep_beams
      select case (prt%status)
      case (PRT_INCOMING, PRT_OUTGOING, PRT_RESONANT)
         flag = .true.
      case (PRT_BEAM)
         flag = kb
      case default
         flag = .false.
      end select
```

```
    end function particle_is_real
```

Polarization status.

⟨Particles: public⟩+≡
```
    public :: particle_get_polarization_status
```

⟨Particles: procedures⟩+≡
```
    elemental function particle_get_polarization_status (prt) result (status)
      integer :: status
      type(particle_t), intent(in) :: prt
      status = prt%polarization
    end function particle_get_polarization_status
```

Return the PDG code from the flavor component directly.

⟨Particles: public⟩+≡
```
    public :: particle_get_pdg
```

⟨Particles: procedures⟩+≡
```
    elemental function particle_get_pdg (prt) result (pdg)
      integer :: pdg
      type(particle_t), intent(in) :: prt
      pdg = flavor_get_pdg (prt%flv)
    end function particle_get_pdg
```

Return the color and anticolor quantum numbers.

⟨Particles: public⟩+≡
```
    public :: particle_get_color
```

⟨Particles: procedures⟩+≡
```
    pure function particle_get_color (prt) result (col)
      integer, dimension(2) :: col
      type(particle_t), intent(in) :: prt
      col(1) = color_get_col (prt%col)
      col(2) = color_get_acl (prt%col)
    end function particle_get_color
```

Return the polarization density matrix (as a shallow copy).

⟨Particles: public⟩+≡
```
    public :: particle_get_polarization
```

⟨Particles: procedures⟩+≡
```
    function particle_get_polarization (prt) result (pol)
      type(polarization_t) :: pol
      type(particle_t), intent(in) :: prt
      pol = prt%pol
    end function particle_get_polarization
```

Return the helicity (if defined and diagonal).

⟨Particles: public⟩+≡
```
    public :: particle_get_helicity
```

916

⟨*Particles: procedures*⟩+≡
```
elemental function particle_get_helicity (prt) result (hel)
  integer :: hel
  integer, dimension(2) :: hel_arr
  type(particle_t), intent(in) :: prt
  hel = 0
  if (helicity_is_defined (prt%hel) .and. &
      helicity_is_diagonal (prt%hel)) then
         hel_arr = helicity_get (prt%hel)
         hel = hel_arr (1)
  end if
end function particle_get_helicity
```

Return the number of children/parents
⟨*Particles: public*⟩+≡
```
public :: particle_get_n_parents
public :: particle_get_n_children
```
⟨*Particles: procedures*⟩+≡
```
elemental function particle_get_n_parents (prt) result (n)
  integer :: n
  type(particle_t), intent(in) :: prt
  if (allocated (prt%parent)) then
     n = size (prt%parent)
  else
     n = 0
  end if
end function particle_get_n_parents

elemental function particle_get_n_children (prt) result (n)
  integer :: n
  type(particle_t), intent(in) :: prt
  if (allocated (prt%child)) then
     n = size (prt%child)
  else
     n = 0
  end if
end function particle_get_n_children
```

Return the array of parents/children.
⟨*Particles: public*⟩+≡
```
public :: particle_get_parents
public :: particle_get_children
```
⟨*Particles: procedures*⟩+≡
```
function particle_get_parents (prt) result (parent)
  type(particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: parent
  if (allocated (prt%parent)) then
     allocate (parent (size (prt%parent)))
     parent = prt%parent
  else
     allocate (parent (0))
  end if
```

```
      end function particle_get_parents

  function particle_get_children (prt) result (child)
    type(particle_t), intent(in) :: prt
    integer, dimension(:), allocatable :: child
    if (allocated (prt%child)) then
       allocate (child (size (prt%child)))
       child = prt%child
    else
       allocate (child (0))
    end if
  end function particle_get_children
```

Return momentum and momentum squared.

⟨*Particles: public*⟩+≡
```
    public :: particle_get_momentum
    public :: particle_get_p2
```

⟨*Particles: procedures*⟩+≡
```
    elemental function particle_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(particle_t), intent(in) :: prt
      p = prt%p
    end function particle_get_momentum

    elemental function particle_get_p2 (prt) result (p2)
      real(default) :: p2
      type(particle_t), intent(in) :: prt
      p2 = prt%p2
    end function particle_get_p2
```

8.2.2 Particle sets

A particle set is what is usually called an event: an array of particles. The
individual particle entries carry momentum, quantum numbers, polarization,
and optionally connections. There is (also optionally) a correlated state-density
matrix that maintains spin correlations that are lost in the individual particle
entries.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_t
```

⟨*Particles: types*⟩+≡
```
    type :: particle_set_t
       ! private
       integer :: n_beam = 0
       integer :: n_in  = 0
       integer :: n_vir = 0
       integer :: n_out = 0
       integer :: n_tot = 0
       type(particle_t), dimension(:), allocatable :: prt
       type(state_matrix_t) :: correlated_state
    end type particle_set_t
```

A particle set can be initialized from an interaction or from a HepMC event record.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_init
```

⟨*Particles: interfaces*⟩+≡
```
    interface particle_set_init
       module procedure particle_set_init_interaction
       module procedure particle_set_init_hepmc
    end interface
```

When a particle set is initialized from a given interaction, we have to determine the branch within the original state matrix that fixes the particle quantum numbers. This is done with the appropriate probabilities, based on a random number `x`. The `mode` determines whether the individual particles become unpolarized, or take a definite (diagonal) helicity, or acquire single-particle polarization matrices. The flag `keep_correlations` tells whether the spin-correlation matrix is to be calculated and stored in addition to the particles. The flag `keep_virtual` tells whether virtual particles should be dropped. Note that if virtual particles are dropped, the spin-correlation matrix makes no sense, and parent-child relations are not set.

For a correct disentangling of color and flavor (in the presence of helicity), we consider two interactions. `int` has no color information, and is used to select a flavor state. Consequently, we trace over helicities here. `int_flows` contains color-flow and potentially helicity information, but is useful only after the flavor combination has been chosen. So this interaction is used to select helicity and color, but restricted to the selected flavor combination.

`int` and `int_flows` may be identical if there is only a single (or no) color flow. If there is just a single flavor combination, `x(1)` can be set to zero.

The current algorithm of evaluator convolution requires that the beam particles are assumed outgoing (in the beam interaction) and become virtual in all derived interactions. In the particle set they should be re-identified as incoming. The optional integer `n_incoming` can be used to perform this correction.

The flag `is_valid` is false if factorization of the state is not possible, in particular if the squared matrix element is zero.

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_init_interaction &
         (particle_set, is_valid, int, int_flows, mode, x, &
          keep_correlations, keep_virtual, n_incoming)
      type(particle_set_t), intent(out) :: particle_set
      logical, intent(out) :: is_valid
      type(interaction_t), intent(in), target :: int, int_flows
      integer, intent(in) :: mode
      real(default), dimension(2), intent(in) :: x
      logical, intent(in) :: keep_correlations, keep_virtual
      integer, intent(in), optional :: n_incoming
      type(state_matrix_t), dimension(:), allocatable, target :: flavor_state
      type(state_matrix_t), dimension(:), allocatable, target :: single_state
      integer :: n_in, n_vir, n_out, n_tot
      type(quantum_numbers_t), dimension(:,:), allocatable :: qn
      logical :: ok
      integer :: i, j
```

```
if (present (n_incoming)) then
   n_in  = n_incoming
   n_vir = interaction_get_n_vir (int) - n_incoming
else
   n_in  = interaction_get_n_in  (int)
   n_vir = interaction_get_n_vir (int)
end if
n_out = interaction_get_n_out (int)
n_tot = interaction_get_n_tot (int)
particle_set%n_in  = n_in
particle_set%n_out = n_out
if (keep_virtual) then
   particle_set%n_vir = n_vir
   particle_set%n_tot = n_tot
else
   particle_set%n_vir = 0
   particle_set%n_tot = n_in + n_out
end if
call interaction_factorize &
     (int, FM_IGNORE_HELICITY, x(1), is_valid, flavor_state)
allocate (qn (n_tot,1))
do i = 1, n_tot
   qn(i,:) = state_matrix_get_quantum_numbers (flavor_state(i), 1)
end do
if (keep_correlations .and. keep_virtual) then
   call interaction_factorize (int_flows, mode, x(2), ok, &
        single_state, particle_set%correlated_state, qn(:,1))
else
   call interaction_factorize (int_flows, mode, x(2), ok, &
        single_state, qn_in=qn(:,1))
end if
is_valid = is_valid .and. ok
allocate (particle_set%prt (particle_set%n_tot))
j = 1
do i = 1, n_tot
   if (i <= n_in) then
      call particle_init &
           (particle_set%prt(j), single_state(i), PRT_INCOMING, mode)
      call particle_set_momentum &
           (particle_set%prt(j), interaction_get_momentum (int, i))
   else if (i <= n_in + n_vir) then
      if (.not. keep_virtual)  cycle
      call particle_init &
           (particle_set%prt(j), single_state(i), PRT_VIRTUAL, mode)
      call particle_set_momentum &
           (particle_set%prt(j), interaction_get_momentum (int, i))
   else
      call particle_init &
           (particle_set%prt(j), single_state(i), PRT_OUTGOING, mode)
      call particle_set_momentum &
           (particle_set%prt(j), interaction_get_momentum (int, i), &
           on_shell = .true.)
   end if
   if (keep_virtual) then
```

```
         call particle_set_children &
              (particle_set%prt(j), interaction_get_children (int, i))
         call particle_set_parents &
              (particle_set%prt(j), interaction_get_parents (int, i))
      end if
      j = j + 1
   end do
   if (keep_virtual) then
      call particle_set_resonance_flag &
           (particle_set%prt, interaction_get_resonance_flags (int))
   end if
   call state_matrix_final (flavor_state)
   call state_matrix_final (single_state)
 end subroutine particle_set_init_interaction
```

If a particle set is initialized from a HepMC event record, we have to specify the treatment of polarization (unpolarized or density matrix) which is common to all particles. Correlated polarization information is not available.

⟨*Particles: procedures*⟩+≡
```
subroutine particle_set_init_hepmc &
     (particle_set, evt, model, polarization)
  type(particle_set_t), intent(out) :: particle_set
  type(hepmc_event_t), intent(in) :: evt
  type(model_t), intent(in), target :: model
  integer, intent(in) :: polarization
  type(hepmc_event_particle_iterator_t) :: it
  type(hepmc_particle_t) :: prt
  integer, dimension(:), allocatable :: barcode
  integer :: n_tot, i
  n_tot = 0
  call hepmc_event_particle_iterator_init (it, evt)
  do while (hepmc_event_particle_iterator_is_valid (it))
     n_tot = n_tot + 1
     call hepmc_event_particle_iterator_advance (it)
  end do
  allocate (barcode (n_tot))
  call hepmc_event_particle_iterator_reset (it)
  do i = 1, n_tot
     barcode(i) = hepmc_particle_get_barcode &
          (hepmc_event_particle_iterator_get (it))
     call hepmc_event_particle_iterator_advance (it)
  end do
  allocate (particle_set%prt (n_tot))
  call hepmc_event_particle_iterator_reset (it)
  do i = 1, n_tot
     prt = hepmc_event_particle_iterator_get (it)
     call particle_init (particle_set%prt(i), &
          prt, model, polarization, barcode)
     call hepmc_event_particle_iterator_advance (it)
  end do
  call hepmc_event_particle_iterator_final (it)
  particle_set%n_tot = n_tot
  particle_set%n_beam  = &
```

```
              count (particle_get_status (particle_set%prt) == PRT_BEAM)
      particle_set%n_in  = &
              count (particle_get_status (particle_set%prt) == PRT_INCOMING)
      particle_set%n_out = &
              count (particle_get_status (particle_set%prt) == PRT_OUTGOING)
      particle_set%n_vir = &
              particle_set%n_tot - particle_set%n_in - particle_set%n_out
    end subroutine particle_set_init_hepmc
```

Manually set the model for the stored particles.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_set_model
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_set_model (particle_set, model)
      type(particle_set_t), intent(inout) :: particle_set
      type(model_t), intent(in), target :: model
      integer :: i
      do i = 1, particle_set%n_tot
         call particle_set_model (particle_set%prt(i), model)
      end do
    end subroutine particle_set_set_model
```

Pointer components are hidden inside the particle polarization, and in the cor-
related state matrix.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_final
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_final (particle_set)
      type(particle_set_t), intent(inout) :: particle_set
      if (allocated (particle_set%prt)) then
         call particle_final (particle_set%prt)
         deallocate(particle_set%prt)
      end if
      call state_matrix_final (particle_set%correlated_state)
    end subroutine particle_set_final
```

Output (default format)

⟨*Particles: public*⟩+≡
```
    public :: particle_set_write
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_write (particle_set, unit, testflag)
      type(particle_set_t), intent(in) :: particle_set
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)") "Particle set:"
      call write_separator (u)
      if (particle_set%n_tot /= 0) then
         do i = 1, particle_set%n_tot
            write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
```

```
             call particle_write (particle_set%prt(i), u, testflag)
          end do
          if (state_matrix_is_defined (particle_set%correlated_state)) then
             call write_separator (u)
             write (u, *) "Correlated state density matrix:"
             call state_matrix_write (particle_set%correlated_state, u)
          end if
       else
          write (u, "(3x,A)") "[empty]"
       end if
    end subroutine particle_set_write
```

8.2.3 I/O formats

Here, we define input/output of particle sets in various formats. This is the right place since particle sets contain most of the event information.

All write/read routines take as first argument the object, as second argument the I/O unit which in this case is a mandatory argument. Then follow further event data.

Internal binary format

This format is supposed to contain the complete information, so the particle data set can be fully reconstructed. The exception is the model part of the particle flavors; this is unassigned for the flavor values read from file.

NOTE: The input routine for a nontrivial state matrix is broken

⟨*Particles: public*⟩+≡
```
    public :: particle_set_write_raw
    public :: particle_set_read_raw
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_write_raw (particle_set, u)
      type(particle_set_t), intent(in) :: particle_set
      integer, intent(in) :: u
      integer :: i
      write (u) &
           particle_set%n_beam, particle_set%n_in, &
           particle_set%n_vir, particle_set%n_out
      write (u) particle_set%n_tot
      do i = 1, particle_set%n_tot
         call particle_write_raw (particle_set%prt(i), u)
      end do
      if (state_matrix_is_defined (particle_set%correlated_state)) then
         call msg_bug ("Write particle set (raw): nontrivial state matrix not supported yet")
      else
         call state_matrix_write_raw (particle_set%correlated_state, u)
      end if
    end subroutine particle_set_write_raw

    subroutine particle_set_read_raw (particle_set, u, iostat)
      type(particle_set_t), intent(out) :: particle_set
      integer, intent(in) :: u
```

```
            integer, intent(out) :: iostat
            integer :: i
            read (u, iostat=iostat) &
                 particle_set%n_beam, particle_set%n_in, &
                 particle_set%n_vir, particle_set%n_out
            read (u, iostat=iostat) particle_set%n_tot
            allocate (particle_set%prt (particle_set%n_tot))
            do i = 1, size (particle_set%prt)
               call particle_read_raw (particle_set%prt(i), u, iostat=iostat)
            end do
            call state_matrix_read_raw (particle_set%correlated_state, u, iostat=iostat)
          end subroutine particle_set_read_raw
```

HepMC format

The master output function fills a HepMC GenEvent object that is already initialized, but has no vertices in it.

We first set up the vertex lists and enter the vertices into the HepMC event. Then, we assign first all incoming particles and then all outgoing particles to their associated vertices. Particles which have neither parent nor children entries (this should not happen) are dropped.

Finally, we insert the beam particles. If there are none, use the incoming particles instead. Transform a particle into a `hepmc_particle` object, including color and polarization. The HepMC status is equivalent to the HEPEVT status, in particular: $0 =$ null entry, $1 =$ physical particle, $2 =$ decayed/fragmented SM hadron, tau or muon, $3 =$ other unphysical particle entry, $4 =$ incoming particles, $11 =$ intermediate resonance such as squarks. The use of 11 for intermediate resonances is as done by HERWIG, see http://herwig.hepforge.org/trac/wiki/FaQs.

⟨*Particles: procedures*⟩+≡
```
          subroutine particle_to_hepmc (prt, hprt)
            type(particle_t), intent(in) :: prt
            type(hepmc_particle_t), intent(out) :: hprt
            integer :: hepmc_status
            select case (particle_get_status (prt))
            case (PRT_UNDEFINED)
               hepmc_status = 0
            case (PRT_OUTGOING)
               hepmc_status = 1
            case (PRT_BEAM)
               hepmc_status = 4
            case (PRT_RESONANT)
               if(abs(particle_get_pdg(prt)) == 13 .or. &
                    abs(particle_get_pdg(prt)) == 15) then
                 hepmc_status = 2
               else
                 hepmc_status = 11
               end if
            case default
               hepmc_status = 3
            end select
            call hepmc_particle_init (hprt, &
```

924

```
                particle_get_momentum (prt), &
                particle_get_pdg (prt), &
                hepmc_status)
       call hepmc_particle_set_color (hprt, &
            particle_get_color (prt))
       select case (particle_get_polarization_status (prt))
       case (PRT_DEFINITE_HELICITY)
          call hepmc_particle_set_polarization (hprt, &
               particle_get_helicity (prt))
       case (PRT_GENERIC_POLARIZATION)
          call hepmc_particle_set_polarization (hprt, &
               particle_get_polarization (prt))
       end select
    end subroutine particle_to_hepmc
```

⟨*Particles: public*⟩+≡
```
    public :: hepmc_event_from_particle_set
```

⟨*Particles: procedures*⟩+≡
```
    subroutine hepmc_event_from_particle_set (evt, particle_set)
      type(hepmc_event_t), intent(inout) :: evt
      type(particle_set_t), intent(in) :: particle_set
      type(hepmc_vertex_t), dimension(:), allocatable :: v
      type(hepmc_particle_t), dimension(:), allocatable :: hprt
      type(hepmc_particle_t), dimension(2) :: hbeam
      logical, dimension(:), allocatable :: is_beam
      integer, dimension(:), allocatable :: v_from, v_to
      integer :: n_vertices, n_tot, i
      n_tot = particle_set%n_tot
      allocate (v_from (n_tot), v_to (n_tot))
      call particle_set_assign_vertices (particle_set, v_from, v_to, n_vertices)
      allocate (v (n_vertices))
      do i = 1, n_vertices
         call hepmc_vertex_init (v(i))
         call hepmc_event_add_vertex (evt, v(i))
      end do
      allocate (hprt (n_tot))
      do i = 1, n_tot
         if (v_to(i) /= 0 .or. v_from(i) /= 0) then
            call particle_to_hepmc (particle_set%prt(i), hprt(i))
         end if
      end do
      allocate (is_beam (n_tot))
      is_beam = particle_get_status (particle_set%prt(1:n_tot)) == PRT_BEAM
      if (.not. any (is_beam)) then
         is_beam = particle_get_status (particle_set%prt(1:n_tot)) == PRT_INCOMING
      end if
      if (count (is_beam) == 2) then
         hbeam = pack (hprt, is_beam)
         call hepmc_event_set_beam_particles (evt, hbeam(1), hbeam(2))
      end if
      do i = 1, n_tot
         if (v_to(i) /= 0) then
            call hepmc_vertex_add_particle_in (v(v_to(i)), hprt(i))
```

```
         end if
      end do
      do i = 1, n_tot
         if (v_from(i) /= 0) then
            call hepmc_vertex_add_particle_out (v(v_from(i)), hprt(i))
         end if
      end do
   end subroutine hepmc_event_from_particle_set
```

Get contents

Find parents/children of a particular particle recursively; the search terminates
if a parent/child has status BEAM, INCOMING, OUTGOING or RESONANT.

⟨*Particles: procedures*⟩+≡

```
   function particle_set_get_real_parents (pset, i, keep_beams) result (parent)
     integer, dimension(:), allocatable :: parent
     type(particle_set_t), intent(in) :: pset
     integer, intent(in) :: i
     logical, intent(in), optional :: keep_beams
     logical, dimension(:), allocatable :: is_real
     logical, dimension(:), allocatable :: is_parent, is_real_parent
     logical :: kb
     integer :: j, k
     kb = .false.
     if (present (keep_beams)) kb = keep_beams
     allocate (is_real (pset%n_tot))
     is_real = particle_is_real (pset%prt, kb)
     allocate (is_parent (pset%n_tot), is_real_parent (pset%n_tot))
     is_real_parent = .false.
     is_parent = .false.
     is_parent(particle_get_parents(pset%prt(i))) = .true.
     do while (any (is_parent))
        where (is_real .and. is_parent)
           is_real_parent = .true.
           is_parent = .false.
        end where
        mark_next_parent: do j = size (is_parent), 1, -1
           if (is_parent(j)) then
              is_parent(particle_get_parents(pset%prt(j))) = .true.
              is_parent(j) = .false.
              exit mark_next_parent
           end if
        end do mark_next_parent
     end do
     allocate (parent (count (is_real_parent)))
     j = 0
     do k = 1, size (is_parent)
        if (is_real_parent(k)) then
           j = j + 1
           parent(j) = k
        end if
     end do
```

```
      end function particle_set_get_real_parents

  function particle_set_get_real_children (pset, i, keep_beams) result (child)
    integer, dimension(:), allocatable :: child
    type(particle_set_t), intent(in) :: pset
    integer, intent(in) :: i
    logical, dimension(:), allocatable :: is_real
    logical, dimension(:), allocatable :: is_child, is_real_child
    logical, intent(in), optional :: keep_beams
    integer :: j, k
    logical :: kb
    kb = .false.
    if (present (keep_beams)) kb = keep_beams
    allocate (is_real (pset%n_tot))
    is_real = particle_is_real (pset%prt, kb)
    allocate (is_child (pset%n_tot), is_real_child (pset%n_tot))
    is_real_child = .false.
    is_child = .false.
    is_child(particle_get_children(pset%prt(i))) = .true.
    do while (any (is_child))
       where (is_real .and. is_child)
          is_real_child = .true.
          is_child = .false.
       end where
       mark_next_child: do j = 1, size (is_child)
          if (is_child(j)) then
             is_child(particle_get_children(pset%prt(j))) = .true.
             is_child(j) = .false.
             exit mark_next_child
          end if
       end do mark_next_child
    end do
    allocate (child (count (is_real_child)))
    j = 0
    do k = 1, size (is_child)
       if (is_real_child(k)) then
          j = j + 1
          child(j) = k
       end if
    end do
  end function particle_set_get_real_children
```

Get the **n_tot**, **n_in**, and **n_out** values out of the particle set.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_get_n_beam
    public :: particle_set_get_n_in
    public :: particle_set_get_n_vir
    public :: particle_set_get_n_out
    public :: particle_set_get_n_tot
```

⟨*Particles: procedures*⟩+≡
```
    function particle_set_get_n_beam (pset) result (n_beam)
       type(particle_set_t), intent(in) :: pset
       integer :: n_beam
```

```
      n_beam = pset%n_beam
    end function particle_set_get_n_beam

    function particle_set_get_n_in (pset) result (n_in)
      type(particle_set_t), intent(in) :: pset
      integer :: n_in
      n_in = pset%n_in
    end function particle_set_get_n_in

    function particle_set_get_n_vir (pset) result (n_vir)
      type(particle_set_t), intent(in) :: pset
      integer :: n_vir
      n_vir = pset%n_in
     end function particle_set_get_n_vir

    function particle_set_get_n_out (pset) result (n_out)
      type(particle_set_t), intent(in) :: pset
      integer :: n_out
      n_out = pset%n_out
    end function particle_set_get_n_out

    function particle_set_get_n_tot (pset) result (n_tot)
      type(particle_set_t), intent(in) :: pset
      integer :: n_tot
      n_tot = pset%n_tot
    end function particle_set_get_n_tot
```

Return a pointer to the particle corresponding to the number
⟨*Particles: public*⟩+≡
```
    public :: particle_set_get_particle
```

⟨*Particles: procedures*⟩+≡
```
    function particle_set_get_particle(pset, index) result(particle)
      type(particle_set_t), intent(in) :: pset
      integer, intent(in) :: index
      type(particle_t) :: particle

      particle = pset%prt(index)
    end function particle_set_get_particle
```

Tools

Given a subevent, reset status codes. If the new status is beam, incoming, or
outgoing, we also make sure that the stored p^2 value is equal to the on-shell
mass squared.
⟨*Particles: public*⟩+≡
```
    public :: particle_set_reset_status
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_reset_status (particle_set, index, status)
      type(particle_set_t), intent(inout) :: particle_set
      integer, dimension(:), intent(in) :: index
      integer, intent(in) :: status
```

```
      integer :: i
      if (allocated (particle_set%prt)) then
         do i = 1, size (index)
            call particle_reset_status (particle_set%prt(index(i)), status)
         end do
      end if
      particle_set%n_beam  = &
           count (particle_get_status (particle_set%prt) == PRT_BEAM)
      particle_set%n_in  = &
           count (particle_get_status (particle_set%prt) == PRT_INCOMING)
      particle_set%n_out = &
           count (particle_get_status (particle_set%prt) == PRT_OUTGOING)
      particle_set%n_vir = particle_set%n_tot &
           - particle_set%n_beam - particle_set%n_in - particle_set%n_out
    end subroutine particle_set_reset_status
```

Reduce a particle set to the essential entries. The entries kept are those with
status INCOMING, OUTGOING or RESONANT. BEAM is kept if keep_beams is true.
Other entries are skipped. The correlated state matrix, if any, is also ignored.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_reduce
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_reduce (pset_in, pset_out, keep_beams)
      type(particle_set_t), intent(in) :: pset_in
      type(particle_set_t), intent(out) :: pset_out
      logical, intent(in), optional :: keep_beams
      integer, dimension(:), allocatable :: status, map
      integer :: i, j
      logical :: kb
      kb = .false.;  if (present (keep_beams))  kb = keep_beams
      allocate (status (pset_in%n_tot))
      status = particle_get_status (pset_in%prt)
      if (kb)  pset_out%n_beam  = count (status == PRT_BEAM)
      pset_out%n_in  = count (status == PRT_INCOMING)
      pset_out%n_vir = count (status == PRT_RESONANT)
      pset_out%n_out = count (status == PRT_OUTGOING)
      pset_out%n_tot = &
           pset_out%n_beam + pset_out%n_in + pset_out%n_vir + pset_out%n_out
      allocate (pset_out%prt (pset_out%n_tot))
      allocate (map (pset_in%n_tot))
      map = 0
      j = 0
      if (kb) call copy_particles (PRT_BEAM)
      call copy_particles (PRT_INCOMING)
      call copy_particles (PRT_RESONANT)
      call copy_particles (PRT_OUTGOING)
      do i = 1, pset_in%n_tot
         if (map(i) == 0)  cycle
         !!! !!! triggers nagfor bug!
         !!! call particle_set_parents (pset_out%prt(map(i)), &
         !!!     map (particle_set_get_real_parents (pset_in, i)))
         !!! call particle_set_children (pset_out%prt(map(i)), &
         !!!     map (particle_set_get_real_children (pset_in, i)))
```

```
      !!! !!! workaround:
      call particle_set_parents (pset_out%prt(map(i)), &
           particle_set_get_real_parents (pset_in, i, kb))
      call particle_set_parents (pset_out%prt(map(i)), &
           map (pset_out%prt(map(i))%parent))
      call particle_set_children (pset_out%prt(map(i)), &
           particle_set_get_real_children (pset_in, i, kb))
      call particle_set_children (pset_out%prt(map(i)), &
           map (pset_out%prt(map(i))%child))
    end do
  contains
    subroutine copy_particles (stat)
      integer, intent(in) :: stat
      integer :: i
      do i = 1, pset_in%n_tot
         if (status(i) == stat) then
            j = j + 1
            map(i) = j
            call particle_init (pset_out%prt(j), pset_in%prt(i))
         end if
      end do
    end subroutine copy_particles
  end subroutine particle_set_reduce
```

Remove the beam particles and beam remnants from the particle set if the keep
beams flag is false. If keep beams is not given, the beam particles and the beam
remnants are removed. The correlated state matrix, if any, is also ignored.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_apply_keep_beams
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_apply_keep_beams (pset_in, pset_out, keep_beams)
      type(particle_set_t), intent(in) :: pset_in
      type(particle_set_t), intent(out) :: pset_out
      logical, intent(in), optional :: keep_beams
      integer, dimension(:), allocatable :: status, map
      integer :: i, j
      logical :: kb
      kb = .false.;  if (present (keep_beams))  kb = keep_beams
      allocate (status (pset_in%n_tot))
      status = particle_get_status (pset_in%prt)
      if (kb)  pset_out%n_beam  = count (status == PRT_BEAM)
      pset_out%n_in  = count (status == PRT_INCOMING)
      if (kb) then
         pset_out%n_vir = count (status == PRT_VIRTUAL) + count (status == PRT_RESONANT) &
              + count (status == PRT_BEAM_REMNANT)
      else
         pset_out%n_vir = count (status == PRT_VIRTUAL) + count (status == PRT_RESONANT)
      end if
      pset_out%n_out = count (status == PRT_OUTGOING)
      pset_out%n_tot = &
           pset_out%n_beam + pset_out%n_in + pset_out%n_vir + pset_out%n_out
      allocate (pset_out%prt (pset_out%n_tot))
      allocate (map (pset_in%n_tot))
```

```
      map = 0
      j = 0
      if (kb) call copy_particles (PRT_BEAM)
      call copy_particles (PRT_INCOMING)
      if (kb) call copy_particles (PRT_BEAM_REMNANT)
      call copy_particles (PRT_RESONANT)
      call copy_particles (PRT_VIRTUAL)
      call copy_particles (PRT_OUTGOING)
      do i = 1, pset_in%n_tot
         if (map(i) == 0)  cycle
         !!! !!! triggers nagfor bug!
         !!!  call particle_set_parents (pset_out%prt(map(i)), &
         !!!       map (particle_set_get_real_parents (pset_in, i)))
         !!!  call particle_set_children (pset_out%prt(map(i)), &
         !!!       map (particle_set_get_real_children (pset_in, i)))
         !!! !!! workaround:
         call particle_set_parents (pset_out%prt(map(i)), &
              particle_set_get_real_parents (pset_in, i, kb))
         call particle_set_parents (pset_out%prt(map(i)), &
              map (pset_out%prt(map(i))%parent))
         call particle_set_children (pset_out%prt(map(i)), &
              particle_set_get_real_children (pset_in, i, kb))
         call particle_set_children (pset_out%prt(map(i)), &
              map (pset_out%prt(map(i))%child))
      end do
    contains
      subroutine copy_particles (stat)
        integer, intent(in) :: stat
        integer :: i
        do i = 1, pset_in%n_tot
           if (status(i) == stat) then
              j = j + 1
              map(i) = j
              call particle_init (pset_out%prt(j), pset_in%prt(i))
           end if
        end do
      end subroutine copy_particles
    end subroutine particle_set_apply_keep_beams
```

Transform a particle set into HEPEVT-compatible form. In this form, for each
particle, the parents and the children are contiguous in the particle array. Usu-
ally, this requires to clone some particles.

We do not know in advance how many particles the canonical form will have.
To be on the safe side, allocate four times the original size.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_to_hepevt_form
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_to_hepevt_form (pset_in, pset_out, keep_beams)
      type(particle_set_t), intent(in) :: pset_in
      type(particle_set_t), intent(out) :: pset_out
      logical, intent(in), optional :: keep_beams
      type(particle_set_t) :: pset
      type :: particle_entry_t
```

```
      integer :: src = 0
      integer :: status = 0
      integer :: orig = 0
      integer :: copy = 0
   end type particle_entry_t
   type(particle_entry_t), dimension(:), allocatable :: prt
   integer, dimension(:), allocatable :: map1, map2
   integer, dimension(:), allocatable :: parent, child
   integer :: n_tot, n_parents, n_children, i, j, c, n

   call particle_set_apply_keep_beams(pset_in, pset, keep_beams)
   n_tot = pset%n_tot
   allocate (prt (4 * n_tot))
   allocate (map1(4 * n_tot))
   allocate (map2(4 * n_tot))
   map1 = 0
   map2 = 0
   allocate (child (n_tot))
   allocate (parent (n_tot))
   n = 0
   do i = 1, n_tot
      if (particle_get_n_parents (pset%prt(i)) == 0) then
         call append (i)
      end if
   end do
   do i = 1, n_tot
      n_children = particle_get_n_children (pset%prt(i))
      if (n_children > 0) then
         child(1:n_children) = particle_get_children (pset%prt(i))
         c = child(1)
         if (map1(c) == 0) then
            n_parents = particle_get_n_parents (pset%prt(c))
            if (n_parents > 1) then
               parent(1:n_parents) = particle_get_parents (pset%prt(c))
               if (i == parent(1) .and. &
                  any( (/(map1(i)+j-1, j=1,n_parents)/) /= map1(parent(1:n_parents)))) &
                  then
                  do j = 1, n_parents
                     call append (parent(j))
                  end do
               end if
            else if (map1(i) == 0) then
               call append (i)
            end if
            do j = 1, n_children
               call append (child(j))
            end do
         end if
      else if (map1(i) == 0) then
         call append (i)
      end if
   end do
   do i = n, 1, -1
      if (prt(i)%status /= PRT_OUTGOING) then
```

```
          do j = 1, i-1
             if (prt(j)%status == PRT_OUTGOING) then
                call append(prt(j)%src)
             end if
          end do
          exit
       end if
    end do
    pset_out%n_beam = count (prt(1:n)%status == PRT_BEAM)
    pset_out%n_in   = count (prt(1:n)%status == PRT_INCOMING)
    pset_out%n_vir  = count (prt(1:n)%status == PRT_RESONANT)
    pset_out%n_out  = count (prt(1:n)%status == PRT_OUTGOING)
    pset_out%n_tot = n
    allocate (pset_out%prt (n))
    do i = 1, n
       call particle_init (pset_out%prt(i), pset%prt(prt(i)%src))
       call particle_reset_status (pset_out%prt(i), prt(i)%status)
       if (prt(i)%orig == 0) then
          !!! !!! This causes nagfor 5.2 (770) Panic
          !!!  call particle_set_parents &
          !!!        (pset_out%prt(i), &
          !!!         map2 (particle_get_parents (pset%prt(prt(i)%src))))
          !!! !!! Workaround
          n_parents = particle_get_n_parents (pset%prt(prt(i)%src))
          parent(1:n_parents) = particle_get_parents (pset%prt(prt(i)%src))
          call particle_set_parents (pset_out%prt(i), &
                                     map2(parent(1:n_parents)))
       else
          call particle_set_parents (pset_out%prt(i), (/ prt(i)%orig /))
       end if
       if (prt(i)%copy == 0) then
          !!! !!! This causes nagfor 5.2 (770) Panic
          !!!  call particle_set_children &
          !!!        (pset_out%prt(i), &
          !!!         map1 (particle_get_children (pset%prt(prt(i)%src))))
          !!! !!! Workaround
          n_children = particle_get_n_children (pset%prt(prt(i)%src))
          child(1:n_children) = particle_get_children (pset%prt(prt(i)%src))
          call particle_set_children (pset_out%prt(i), &
                                      map1(child(1:n_children)))
       else
          call particle_set_children (pset_out%prt(i), (/ prt(i)%copy /))
       end if
    end do
  contains
    subroutine append (i)
      integer, intent(in) :: i
      n = n + 1
      if (n > size (prt)) &
          call msg_bug ("Particle set transform to HEPEVT: insufficient space")
      prt(n)%src = i
      prt(n)%status = particle_get_status (pset%prt(i))
      if (map1(i) == 0) then
        map1(i) = n
```

```
    else
       prt(map2(i))%status = PRT_VIRTUAL
       prt(map2(i))%copy = n
       prt(n)%orig = map2(i)
    end if
    map2(i) = n
  end subroutine append
 end subroutine particle_set_to_hepevt_form
```

Reconstruct the momenta within an interaction object from a particle set. We start from the incoming particles, which should be stored at the beginning. (The number of incoming particles is provided separately, so we do not need to analyze the interaction or particle set for this.) We follow their children down the tree until the interaction is exhausted.

This implies that the particle set may contain a longer tree (e.g., decay chains) than the interaction. The remainder of the tree is simply ignored.

The detailed ordering of the subsequent sub-interactions may differ between the interaction and the particle set. However, within each sub-interaction, the ordering of particles must match.

It may happen that the particle set lacks beam and remnant particles which are present in the interaction. In this case, the beam particles (the first two entries in the interaction) will not be filled. However, we can use the source link of the beam particles to recover their momenta from the actual beam interaction (which is stored somewhere else), and then recover the beam-remnant momenta as the difference of the beam and parton momenta.

⟨*Particles: public*⟩+≡
```
   public :: particle_set_fill_interaction
```

⟨*Particles: procedures*⟩+≡
```
   subroutine particle_set_fill_interaction (pset, int, n_in, recover_beams)
     type(particle_set_t), intent(in) :: pset
     type(interaction_t), intent(inout) :: int
     integer, intent(in) :: n_in
     logical, intent(in), optional :: recover_beams
     logical, dimension(:), allocatable :: p_is_set
     logical :: recover
     integer, dimension(:), allocatable :: i_parent, j_parent
     integer :: n_tot, k
     n_tot = interaction_get_n_tot (int)
     recover = .false.;  if (present (recover_beams))  recover = recover_beams
     allocate (p_is_set (n_tot), source = .false.)
     allocate (i_parent (n_in), source = [(k, k = 1, n_in)])
     allocate (j_parent (n_in), source = i_parent)
     do k = 1, n_in
        call fill_subint (i_parent(k), j_parent(k))
     end do
     if (recover) then
        do k = 1, n_in
           call recover_beam_and_remnant (k)
        end do
     end if
     if (.not. all (p_is_set)) then
        call particle_set_write (pset)
```

```
      call interaction_write (int)
      call msg_fatal ("Mismatch between particle set and interaction")
    end if
  contains
    recursive subroutine fill_subint (i, j)
      integer, intent(in) :: i, j
      integer, dimension(:), allocatable :: i_child, j_child
      integer :: n_child, n_child_pset, k
      if (p_is_set(i))  return
      call interaction_set_momentum (int, &
          particle_get_momentum (pset%prt(j)), i)
      p_is_set(i) = .true.
      n_child = interaction_get_n_children (int, i)
      n_child_pset = particle_get_n_children (pset%prt(j))
      if (n_child /= 0 .and. n_child == n_child_pset) then
         allocate (i_child (n_child))
         allocate (j_child (n_child))
         i_child = interaction_get_children (int, i)
         j_child = particle_get_children (pset%prt(j))
         do k = 1, n_child
            call fill_subint (i_child(k), j_child(k))
         end do
      end if
    end subroutine fill_subint
    subroutine recover_beam_and_remnant (k)
      integer, intent(in) :: k
      integer :: k_src, k_in, k_rem
      type(interaction_t), pointer :: int_src
      integer, dimension(2) :: i_child
      call interaction_find_source (int, k, int_src, k_src)
      call interaction_set_momentum (int, &
          interaction_get_momentum (int_src, k_src), k)
      i_child = interaction_get_children (int, k)
      if (interaction_get_n_children (int, i_child(1)) > 0) then
         k_in = i_child(1);  k_rem = i_child(2)
      else
         k_in = i_child(2);  k_rem = i_child(1)
      end if
      call interaction_set_momentum (int, &
          interaction_get_momentum (int, k) &
          - interaction_get_momentum (int, k_in), k_rem)
    end subroutine recover_beam_and_remnant
  end subroutine particle_set_fill_interaction
```

This reconstructs the hard interaction from a particle set. First the particle set
is reduced to the incoming partons, then we add the direct children of those.
The interaction and particle set should match in their array sizes. If we find a
permutation of the particle flavors that matches one of the given flavor states,
the assignment of momenta is made accordingly.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_extract_interaction
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_extract_interaction (pset, int, flv_state)
```

```
      type(particle_set_t), intent(in) :: pset
      type(interaction_t), intent(inout) :: int
      integer, dimension(:,:), intent(in) :: flv_state
      integer :: n_in, n_out, n_tot
      integer, dimension(:), allocatable :: status, incoming, outgoing, index
      integer, dimension(:), allocatable :: pdg, perm
      integer :: i
      logical :: ok
      allocate (status (pset%n_tot))
      status = particle_get_status (pset%prt)
      n_in = count (status == PRT_INCOMING)
      allocate (incoming (n_in))
      incoming = pack ((/ (i, i = 1, pset%n_tot) /), status == PRT_INCOMING)
      i = incoming (1)
      n_out = particle_get_n_children (pset%prt(i))
      allocate (outgoing (n_out))
      outgoing = particle_get_children (pset%prt(i))
      n_tot = n_in + n_out
      if (n_in /= interaction_get_n_in (int) &
           .or. n_out /= interaction_get_n_out (int) &
           .or. n_tot /= interaction_get_n_tot (int)) then
         call msg_fatal &
              ("This event does not match the associated process (size)")
         return
      end if
      allocate (index (n_tot), pdg (n_tot), perm (n_tot))
      index(:n_in) = incoming
      index(n_in+1:) = outgoing
      pdg = particle_get_pdg (pset%prt(index))
      call find_flavor_ordering (flv_state, pdg, n_in, perm, ok)
      if (.not. ok) then
         call particle_set_write (pset)
         call msg_fatal &
              ("This event does not match the associated process (flavors)")
         return
      end if
      do i = 1, n_tot
         call interaction_set_momentum (int, &
              particle_get_momentum (pset%prt(i)), perm(i))
      end do
    end subroutine particle_set_extract_interaction
```

Given a particle set with an arbitrary ordering of particles, we need the permutation that yields an ordering that is present in the state matrix. This can be found by examining the flavor table. The result is a permutation of the PDG array that matches a row in the flavor table. If ok is returned false, no match was found.

⟨*Particles: procedures*⟩+≡

```
    subroutine find_flavor_ordering (flv_state, pdg, n_in, perm, ok)
      integer, dimension(:,:), intent(in) :: flv_state
      integer, dimension(:), intent(in) :: pdg
      integer, intent(in) :: n_in
      integer, dimension(:), intent(out) :: perm
```

```
      logical, intent(out) :: ok
      integer :: n_tot, f, i, j, k
      logical, dimension(:), allocatable :: found
      n_tot = size (pdg)
      if (size (flv_state, 1) /= n_tot) then
         ok = .false.
         return
      end if
      do i = 1, n_in
         perm(i) = i
      end do
      allocate (found (n_tot))
      ok = .false.
      do f = 1, size (flv_state, 2)
         call find_ordering_for_this_state (flv_state(:,f))
      end do
    contains
      subroutine find_ordering_for_this_state (pdg_state)
        integer, dimension(:), intent(in) :: pdg_state
        found = .false.
        if (all (pdg_state(1:n_in) == pdg(1:n_in))) then
           SCAN_INPUT: do j = n_in + 1, n_tot
              SCAN_STATE: do k = n_in + 1, n_tot
                 if (found(k))  cycle SCAN_STATE
                 if (pdg_state(k) == pdg(j)) then
                    found(k) = .true.
                    perm(j) = k
                    cycle SCAN_INPUT
                 end if
              end do SCAN_STATE
              return
           end do SCAN_INPUT
           ok = .true.
        end if
      end subroutine find_ordering_for_this_state
    end subroutine find_flavor_ordering
```

This procedure reconstructs an array of vertex indices from the parent-child information in the particle entries, according to the HepMC scheme. For each particle, we determine which vertex it comes from and which vertex it goes to. We return the two arrays and the maximum vertex index.

For each particle in the list, we first check its parents. If for any parent the vertex where it goes to is already known, this vertex index is assigned as the current 'from' vertex. Otherwise, a new index is created, assigned as the current 'from' vertex, and as the 'to' vertex for all parents.

Then, the analogous procedure is done for the children.

⟨*Particles: public*⟩+≡
```
    public :: particle_set_assign_vertices
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particle_set_assign_vertices &
        (particle_set, v_from, v_to, n_vertices)
      type(particle_set_t), intent(in) :: particle_set
```

```
      integer, dimension(:), intent(out) :: v_from, v_to
      integer, intent(out) :: n_vertices
      integer, dimension(:), allocatable :: parent, child
      integer :: n_parents, n_children, vf, vt
      integer :: i, j, v
      v_from = 0
      v_to = 0
      vf = 0
      vt = 0
      do i = 1, particle_set%n_tot
         n_parents = particle_get_n_parents (particle_set%prt(i))
         if (n_parents /= 0) then
            allocate (parent (n_parents))
            parent = particle_get_parents (particle_set%prt(i))
            SCAN_PARENTS: do j = 1, size (parent)
               v = v_to(parent(j))
               if (v /= 0) then
                  v_from(i) = v;  exit SCAN_PARENTS
               end if
            end do SCAN_PARENTS
            if (v_from(i) == 0) then
               vf = vf + 1;  v_from(i) = vf
               v_to(parent) = vf
            end if
            deallocate (parent)
         end if
         n_children = particle_get_n_children (particle_set%prt(i))
         if (n_children /= 0) then
            allocate (child (n_children))
            child = particle_get_children (particle_set%prt(i))
            SCAN_CHILDREN: do j = 1, size (child)
               v = v_from(child(j))
               if (v /= 0) then
                  v_to(i) = v;  exit SCAN_CHILDREN
               end if
            end do SCAN_CHILDREN
            if (v_to(i) == 0) then
               vt = vt + 1;  v_to(i) = vt
               v_from(child) = vt
            end if
            deallocate (child)
         end if
      end do
      n_vertices = max (vf, vt)
   end subroutine particle_set_assign_vertices
```

8.2.4 Expression interface

This converts a `particle_set` object as defined here to a more concise `subevt` object that can be used as the event root of an expression. In particular, the latter lacks virtual particles, spin correlations and parent-child relations.

⟨*Particles: public*⟩+≡

```
      public :: particle_set_to_subevt
⟨Particles: procedures⟩+≡
    subroutine particle_set_to_subevt (particle_set, subevt)
      type(particle_set_t), intent(in), target :: particle_set
      type(subevt_t), intent(out) :: subevt
      type(particle_t), pointer :: prt
      integer :: i, k
      integer, dimension(2) :: hel
      call subevt_init &
           (subevt, particle_set%n_beam + particle_set%n_in + particle_set%n_out)
      k = 0
      do i = 1, particle_set%n_tot
         prt => particle_set%prt(i)
         select case (particle_get_status (prt))
         case (PRT_BEAM)
            k = k + 1
            call subevt_set_beam (subevt, k, &
                 particle_get_pdg (prt), &
                 particle_get_momentum (prt), &
                 particle_get_p2 (prt))
         case (PRT_INCOMING)
            k = k + 1
            call subevt_set_incoming (subevt, k, &
                 particle_get_pdg (prt), &
                 particle_get_momentum (prt), &
                 particle_get_p2 (prt))
         case (PRT_OUTGOING)
            k = k + 1
            call subevt_set_outgoing (subevt, k, &
                 particle_get_pdg (prt), &
                 particle_get_momentum (prt), &
                 particle_get_p2 (prt))
         end select
         select case (particle_get_status (prt))
         case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
            if (prt%polarization == PRT_DEFINITE_HELICITY) then
               if (helicity_is_diagonal (prt%hel)) then
                  hel = helicity_get (prt%hel)
                  call subevt_polarize (subevt, k, hel(1))
               end if
            end if
         end select
      end do
    end subroutine particle_set_to_subevt
```

This replaces the particle_set

⟨Particles: public⟩+≡
```
      public :: particle_set_replace
```
⟨Particles: procedures⟩+≡
```
    subroutine particle_set_replace (particle_set, newprt)
      type(particle_set_t), intent(inout) :: particle_set
      type(particle_t), intent(in), dimension(:), allocatable :: newprt
      if (allocated (particle_set%prt))  deallocate (particle_set%prt)
```

```
          allocate (particle_set%prt(size (newprt)))
          particle_set%prt = newprt
          particle_set%n_tot = size (newprt)
          particle_set%n_beam = count (particle_get_status (newprt) == PRT_BEAM)
          particle_set%n_in = count (particle_get_status (newprt) == PRT_INCOMING)
          particle_set%n_out = count (particle_get_status (newprt) == PRT_OUTGOING)
          particle_set%n_vir = particle_set%n_tot &
                - particle_set%n_beam - particle_set%n_in - particle_set%n_out
        end subroutine particle_set_replace
```

Eliminate numerical noise

⟨*Particles: public*⟩+≡
```
    public :: pacify
```

⟨*Particles: interfaces*⟩+≡
```
    interface pacify
       module procedure pacify_particle
       module procedure pacify_particle_set
    end interface pacify
```

⟨*Particles: procedures*⟩+≡
```
    subroutine pacify_particle (prt)
      class(particle_t), intent(inout) :: prt
      real(default) :: e
      e = epsilon (1._default) * energy (prt%p)
      call pacify (prt%p, 10 * e)
      call pacify (prt%p2, 1e4 * e)
    end subroutine pacify_particle

    subroutine pacify_particle_set (pset)
      class(particle_set_t), intent(inout) :: pset
      integer :: i
      do i = 1, pset%n_tot
         call pacify (pset%prt(i))
      end do
    end subroutine pacify_particle_set
```

8.2.5 Unit tests

Set up a chain of production and decay and factorize the result into particles.
The process is $d\bar{d} \to Z \to q\bar{q}$.

⟨*Particles: public*⟩+≡
```
    public :: particles_test
```

⟨*Particles: procedures*⟩+≡
```
    subroutine particles_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Particles: execute tests*⟩
```
    end subroutine particles_test
```

Check the basic setup of the `particle_set_t` type.

⟨Particles: execute tests⟩≡
```
    call test (particles_1, "particles_1", &
         "check particle_set routines", &
         u, results)
```

⟨Particles: tests⟩≡
```
    subroutine particles_1 (u)
```
  ⟨Particles: common test code 1⟩

```
      write (u, "(A)")  "* Test output: Particles"
      write (u, "(A)")  "*   Purpose: test particle_set routines"
      write (u, "(A)")
```

  ⟨Particles: common test code 2⟩

```
      write (u, "(A)")
      write (u, "(A)")  &
           "* Factorize (complete, polarized, correlated); write and read again"
      write (u, "(A)")

      int => evaluator_get_int_ptr (eval)
      call particle_set_init &
           (particle_set3, ok, int, int, FM_FACTOR_HELICITY, &
            [0.7_default, 0.7_default], .true., .true.)
      call particle_set_write (particle_set3, u)
```

```
  !!! Raw I/O with state matrices currently disabled, cf. #627.
      !!! unit = free_unit ()
      !!! open (unit, action="readwrite", form="unformatted", status="scratch")
      !!! call particle_set_write_raw (particle_set3, unit)
      !!! rewind (unit)
      !!! call particle_set_read_raw (particle_set4, unit, iostat=iostat)
      !!! close (unit)
      !!!
      !!! write (u, "(A)")
      !!! call particle_set_write (particle_set4, u)
      !!! write (u, "(A)")
      !!! write (u, "(A)")  "* Transform to a subevt object"
      !!! write (u, "(A)")
      !!!
      !!! call particle_set_to_subevt (particle_set4, subevt)
      !!! call subevt_write (subevt, u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call particle_set_final (particle_set1)
      call particle_set_final (particle_set2)
      call particle_set_final (particle_set3)
      call particle_set_final (particle_set4)
      call evaluator_final (eval)
      call interaction_final (int1)
      call interaction_final (int2)
      call model_list%final ()
```

```
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: particles_1"

   end subroutine particles_1
```

If `HepMC` is available, check the routines via `HepMC`.

⟨*Particles: execute tests*⟩+≡
```
      if (hepmc_is_available ()) then
         call test (particles_2, "particles_2", &
              "check particle_set routines via HepMC", &
              u, results)
      end if
```

⟨*Particles: tests*⟩+≡
```
      subroutine particles_2 (u)
   ⟨Particles: common test code 1⟩

      write (u, "(A)")  "* Test output: Particles"
      write (u, "(A)")  "*   Purpose: test particle_set routines via HepMC"
      write (u, "(A)")

   ⟨Particles: common test code 2⟩

      write (u, "(A)")
      write (u, "(A)")  "* Transfer particle_set to HepMC, print, and output to"
      write (u, "(A)")  "         particles_test.hepmc.dat"
      write (u, "(A)")

      call hepmc_event_init (hepmc_event, 11, 127)
      call hepmc_event_from_particle_set (hepmc_event, particle_set2)
      call hepmc_event_print (hepmc_event)
      call hepmc_iostream_open_out &
           (iostream , var_str ("particles_test.hepmc.dat"))
      call hepmc_iostream_write_event (iostream, hepmc_event)
      call hepmc_iostream_close (iostream)

      write (u, "(A)")
      write (u, "(A)")  "* Recover from HepMC file"
      write (u, "(A)")

      call particle_set_final (particle_set2)
      call hepmc_event_final (hepmc_event)
      call hepmc_event_init (hepmc_event)
      call hepmc_iostream_open_in &
           (iostream , var_str ("particles_test.hepmc.dat"))
      call hepmc_iostream_read_event (iostream, hepmc_event, ok)
      call hepmc_iostream_close (iostream)
      call particle_set_init (particle_set2, &
           hepmc_event, model, PRT_DEFINITE_HELICITY)
      call particle_set_write (particle_set2, u)

      write (u, "(A)")
```

```
        write (u, "(A)")  "* Cleanup"

        call particle_set_final (particle_set1)
        call particle_set_final (particle_set2)
        call evaluator_final (eval)
        call interaction_final (int1)
        call interaction_final (int2)
        call hepmc_event_final (hepmc_event)
        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: particles_2"

    end subroutine particles_2
```

⟨*Particles: common test code 1*⟩≡
```
        use os_interface
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_list_t) :: model_list
        type(model_t), pointer :: model => null ()
        type(flavor_t), dimension(3) :: flv
        type(color_t), dimension(3) :: col
        type(helicity_t), dimension(3) :: hel
        type(quantum_numbers_t), dimension(3) :: qn
        type(vector4_t), dimension(3) :: p
        type(interaction_t), target :: int1, int2
        type(quantum_numbers_mask_t) :: qn_mask_conn, qn_rest
        type(evaluator_t), target :: eval
        type(interaction_t), pointer :: int
        type(particle_set_t) :: particle_set1, particle_set2
        type(particle_set_t) :: particle_set3, particle_set4
        type(subevt_t) :: subevt
        type(hepmc_event_t) :: hepmc_event
        type(hepmc_iostream_t) :: iostream
        logical :: ok
        integer :: unit, iostat
```
⟨*Particles: common test code 2*⟩≡
```
        write (u, "(A)")  "* Reading model file"

        call syntax_model_file_init ()
        call os_data_init (os_data)
        call model_list%read_model &
            (var_str("SM"), var_str("SM.mdl"), os_data, model)

        write (u, "(A)")
        write (u, "(A)")  "* Initializing production process"

        call interaction_init (int1, 2, 0, 1, set_relations=.true.)
        call flavor_init (flv, [1, -1, 23], model)
        call helicity_init (hel(3), 1, 1)
        call quantum_numbers_init (qn, flv, hel)
```

```
call interaction_add_state (int1, qn, value=(0.25_default, 0._default))
call helicity_init (hel(3), 1,-1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0._default, 0.25_default))
call helicity_init (hel(3),-1, 1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0._default,-0.25_default))
call helicity_init (hel(3),-1,-1)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.25_default, 0._default))
call helicity_init (hel(3), 0, 0)
call quantum_numbers_init (qn, flv, hel)
call interaction_add_state (int1, qn, value=(0.5_default, 0._default))
call interaction_freeze (int1)
p(1) = vector4_moving (45._default, 45._default, 3)
p(2) = vector4_moving (45._default,-45._default, 3)
p(3) = p(1) + p(2)
call interaction_set_momenta (int1, p)

write (u, "(A)")
write (u, "(A)")  "* Setup decay process"

call interaction_init (int2, 1, 0, 2, set_relations=.true.)
call flavor_init (flv, [23, 1, -1], model)
call color_init_col_acl (col, [0, 501, 0], [0, 0, 501])
call helicity_init (hel, [1, 1, 1], [1, 1, 1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(1._default, 0._default))
call helicity_init (hel, [1, 1, 1], [-1,-1,-1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0._default, 0.1_default))
call helicity_init (hel, [-1,-1,-1], [1, 1, 1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0._default,-0.1_default))
call helicity_init (hel, [-1,-1,-1], [-1,-1,-1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(1._default, 0._default))
call helicity_init (hel, [0, 1,-1], [0, 1,-1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(4._default, 0._default))
call helicity_init (hel, [0,-1, 1], [0, 1,-1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(2._default, 0._default))
call helicity_init (hel, [0, 1,-1], [0,-1, 1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(2._default, 0._default))
call helicity_init (hel, [0,-1, 1], [0,-1, 1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(4._default, 0._default))
call flavor_init (flv, [23, 2, -2], model)
call helicity_init (hel, [0, 1,-1], [0, 1,-1])
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0.5_default, 0._default))
call helicity_init (hel, [0,-1, 1], [0,-1, 1])
```

944

```
call quantum_numbers_init (qn, flv, col, hel)
call interaction_add_state (int2, qn, value=(0.5_default, 0._default))
call interaction_freeze (int2)
p(2) = vector4_moving (45._default, 45._default, 2)
p(3) = vector4_moving (45._default,-45._default, 2)
call interaction_set_momenta (int2, p)
call interaction_set_source_link (int2, 1, int1, 3)
call interaction_write (int1, u)
call interaction_write (int2, u)

write (u, "(A)")
write (u, "(A)")  "* Concatenate production and decay"

call evaluator_init_product (eval, int1, int2, qn_mask_conn, &
     connections_are_resonant=.true.)
call evaluator_receive_momenta (eval)
call eval%evaluate ()
call eval%write (u)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (complete, polarized)"
write (u, "(A)")

int => evaluator_get_int_ptr (eval)
call particle_set_init &
     (particle_set1, ok, int, int, FM_FACTOR_HELICITY, &
      [0.2_default, 0.2_default], .false., .true.)
call particle_set_write (particle_set1, u)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (in/out only, selected helicity)"
write (u, "(A)")

int => evaluator_get_int_ptr (eval)
call particle_set_init &
     (particle_set2, ok, int, int, FM_SELECT_HELICITY, &
      [0.9_default, 0.9_default], .false., .false.)
call particle_set_write (particle_set2, u)
call particle_set_final (particle_set2)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (complete, selected helicity)"
write (u, "(A)")

int => evaluator_get_int_ptr (eval)
call particle_set_init &
     (particle_set2, ok, int, int, FM_SELECT_HELICITY, &
      [0.7_default, 0.7_default], .false., .true.)
call particle_set_write (particle_set2, u)
```

Chapter 9

Initial State

9.1 Beam structure

This module stores the beam structure definition as it is declared in the SIN-DARIN script. The structure definition is not analyzed, just recorded for later use.

We do not capture any numerical parameters, just names of particles and structure functions.

⟨**beam_structures.f90**⟩≡
  ⟨*File header*⟩

  ```
  module beam_structures
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use limits, only: FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests

    use variables
    use polarizations
```

  ⟨*Standard module head*⟩

  ⟨*Beam structures: public*⟩

  ⟨*Beam structures: types*⟩

  ⟨*Beam structures: interfaces*⟩

  ```
  contains
```

  ⟨*Beam structures: procedures*⟩

  ⟨*Beam structures: tests*⟩

```
    end module beam_structures
```

9.1.1   Beam structure elements

An entry in a beam-structure record consists of a string that denotes a type of structure function.

⟨*Beam structures: types*⟩≡
```
    type :: beam_structure_entry_t
      logical :: is_valid = .false.
      type(string_t) :: name
     contains
     ⟨Beam structures: beam structure entry: TBP⟩
    end type beam_structure_entry_t
```

Output.

⟨*Beam structures: beam structure entry: TBP*⟩≡
```
    procedure :: to_string => beam_structure_entry_to_string
```

⟨*Beam structures: procedures*⟩≡
```
    function beam_structure_entry_to_string (object) result (string)
      class(beam_structure_entry_t), intent(in) :: object
      type(string_t) :: string
      if (object%is_valid) then
         string = object%name
      else
         string = "none"
      end if
    end function beam_structure_entry_to_string
```

A record in the beam-structure sequence denotes either a structure-function entry, a pair of such entries, or a pair spectrum.

⟨*Beam structures: types*⟩+≡
```
    type :: beam_structure_record_t
      type(beam_structure_entry_t), dimension(:), allocatable :: entry
    end type beam_structure_record_t
```

9.1.2   Beam structure type

The beam-structure object contains the beam particle(s) as simple strings. The sequence of records indicates the structure functions by name. No numerical parameters are stored.

⟨*Beam structures: public*⟩≡
```
    public :: beam_structure_t
```

⟨*Beam structures: types*⟩+≡
```
    type :: beam_structure_t
      private
      integer :: n_beam = 0
      type(string_t), dimension(:), allocatable :: prt
      type(beam_structure_record_t), dimension(:), allocatable :: record
      type(smatrix_t), dimension(:), allocatable :: smatrix
```

```
      real(default), dimension(:), allocatable :: pol_f
      real(default), dimension(:), allocatable :: p
      real(default), dimension(:), allocatable :: theta
      real(default), dimension(:), allocatable :: phi
    contains
    ⟨Beam structures: beam structure: TBP⟩
    end type beam_structure_t
```

The finalizer deletes all contents explicitly, so we can continue with an empty
beam record. (It is not needed for deallocation.) We have distinct finalizers for
the independent parts of the beam structure.

⟨*Beam structures: beam structure: TBP*⟩≡
```
    procedure :: final_sf => beam_structure_final_sf
```

⟨*Beam structures: procedures*⟩+≡
```
    subroutine beam_structure_final_sf (object)
      class(beam_structure_t), intent(inout) :: object
      if (allocated (object%prt))  deallocate (object%prt)
      if (allocated (object%record))  deallocate (object%record)
      object%n_beam = 0
    end subroutine beam_structure_final_sf
```

Output. The actual information fits in a single line, therefore we can provide a
`to_string` method. The `show` method also lists the current values of relevant
global variables.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: write => beam_structure_write
    procedure :: to_string => beam_structure_to_string
```

⟨*Beam structures: procedures*⟩+≡
```
    subroutine beam_structure_write (object, unit)
      class(beam_structure_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A,A)")  "Beam structure: ", char (object%to_string ())
      if (allocated (object%smatrix)) then
         do i = 1, size (object%smatrix)
            write (u, "(3x,A,I0,A)") "polarization (beam ", i, "):"
            call object%smatrix(i)%write (u, indent=2)
         end do
      end if
      if (allocated (object%pol_f)) then
         write (u, "(3x,A,F10.7,:,',',F10.7)")  "polarization degree =", &
              object%pol_f
      end if
      if (allocated (object%p)) then
         write (u, "(3x,A," // FMT_19 // ",:,','," // FMT_19 // &
              ")")  "momentum =", object%p
      end if
      if (allocated (object%theta)) then
         write (u, "(3x,A," // FMT_19 // ",:,','," // FMT_19 // &
              ")")  "angle th =", object%theta
```

```
         end if
       if (allocated (object%phi)) then
          write (u, "(3x,A," // FMT_19 // ",:,',',"  // FMT_19 // &
               ")")  "angle ph =", object%phi
       end if
    end subroutine beam_structure_write

    function beam_structure_to_string (object, sf_only) result (string)
      class(beam_structure_t), intent(in) :: object
      logical, intent(in), optional :: sf_only
      type(string_t) :: string
      integer :: i, j
      logical :: with_beams
      with_beams = .true.;  if (present (sf_only))  with_beams = .not. sf_only
      select case (object%n_beam)
      case (1)
         if (with_beams) then
            string = object%prt(1)
         else
            string = ""
         end if
      case (2)
         if (with_beams) then
            string = object%prt(1) // ", " // object%prt(2)
         else
            string = ""
         end if
         if (allocated (object%record)) then
            if (size (object%record) > 0) then
               if (with_beams)  string = string // " => "
               do i = 1, size (object%record)
                  if (i > 1)  string = string // " => "
                  do j = 1, size (object%record(i)%entry)
                     if (j > 1)  string = string // ", "
                     string = string // object%record(i)%entry(j)%to_string ()
                  end do
               end do
            end if
         end if
      case default
         string = "[any particles]"
      end select
    end function beam_structure_to_string
```

Initializer: dimension the beam structure record. Each array element denotes
the number of entries for a record within the beam-structure sequence. The
number of entries is either one or two, while the number of records is unlimited.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: init_sf => beam_structure_init_sf
```

⟨*Beam structures: procedures*⟩+≡
```
    subroutine beam_structure_init_sf (beam_structure, prt, dim_array)
      class(beam_structure_t), intent(inout) :: beam_structure
      type(string_t), dimension(:), intent(in) :: prt
```

```
      integer, dimension(:), intent(in), optional :: dim_array
      integer :: i
      call beam_structure%final_sf ()
      beam_structure%n_beam = size (prt)
      allocate (beam_structure%prt (size (prt)))
      beam_structure%prt = prt
      if (present (dim_array)) then
         allocate (beam_structure%record (size (dim_array)))
         do i = 1, size (dim_array)
            allocate (beam_structure%record(i)%entry (dim_array(i)))
         end do
      else
         allocate (beam_structure%record (0))
      end if
   end subroutine beam_structure_init_sf
```

Set an entry, specified by record number and entry number.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
   procedure :: set_sf => beam_structure_set_sf
```

⟨*Beam structures: procedures*⟩+≡
```
   subroutine beam_structure_set_sf (beam_structure, i, j, name)
     class(beam_structure_t), intent(inout) :: beam_structure
     integer, intent(in) :: i, j
     type(string_t), intent(in) :: name
     associate (entry => beam_structure%record(i)%entry(j))
       entry%name = name
       entry%is_valid = .true.
     end associate
   end subroutine beam_structure_set_sf
```

Expand the beam-structure object. (i) For a pair spectrum, keep the entry. (ii) For a single-particle structure function written as a single entry, replace this by a record with two entries. (ii) For a record with two nontrivial entries, separate this into two records with one trivial entry each.

To achieve this, we need a function that tells us whether an entry is a spectrum or a structure function. It returns 0 for a trivial entry, 1 for a single-particle structure function, and 2 for a two-particle spectrum.

⟨*Beam structures: interfaces*⟩≡
```
   abstract interface
     function strfun_mode_fun (name) result (n)
       import
       type(string_t), intent(in) :: name
       integer :: n
     end function strfun_mode_fun
   end interface
```

Algorithm: (1) Mark entries as invalid where necessary. (2) Count the number of entries that we will need. (3) Expand and copy entries to a new record array. (4) Replace the old array by the new one.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
   procedure :: expand => beam_structure_expand
```

⟨*Beam structures: procedures*⟩+≡
```fortran
    subroutine beam_structure_expand (beam_structure, strfun_mode)
      class(beam_structure_t), intent(inout) :: beam_structure
      procedure(strfun_mode_fun) :: strfun_mode
      type(beam_structure_record_t), dimension(:), allocatable :: new
      integer :: n_record, i, j
      if (.not. allocated (beam_structure%record))  return
      do i = 1, size (beam_structure%record)
         associate (entry => beam_structure%record(i)%entry)
           do j = 1, size (entry)
              select case (strfun_mode (entry(j)%name))
              case (0);  entry(j)%is_valid = .false.
              end select
           end do
         end associate
      end do
      n_record = 0
      do i = 1, size (beam_structure%record)
         associate (entry => beam_structure%record(i)%entry)
           select case (size (entry))
           case (1)
              if (entry(1)%is_valid) then
                 select case (strfun_mode (entry(1)%name))
                 case (1);  n_record = n_record + 2
                 case (2);  n_record = n_record + 1
                 end select
              end if
           case (2)
              do j = 1, 2
                 if (entry(j)%is_valid) then
                    select case (strfun_mode (entry(j)%name))
                    case (1);  n_record = n_record + 1
                    case (2)
                       call beam_structure%write ()
                       call msg_fatal ("Pair spectrum used as &
                            &single-particle structure function")
                    end select
                 end if
              end do
           end select
         end associate
      end do
      allocate (new (n_record))
      n_record = 0
      do i = 1, size (beam_structure%record)
         associate (entry => beam_structure%record(i)%entry)
           select case (size (entry))
           case (1)
              if (entry(1)%is_valid) then
                 select case (strfun_mode (entry(1)%name))
                 case (1)
                    n_record = n_record + 1
                    allocate (new(n_record)%entry (2))
                    new(n_record)%entry(1) = entry(1)
```

```
                    n_record = n_record + 1
                    allocate (new(n_record)%entry (2))
                    new(n_record)%entry(2) = entry(1)
                 case (2)
                    n_record = n_record + 1
                    allocate (new(n_record)%entry (1))
                    new(n_record)%entry(1) = entry(1)
                 end select
              end if
           case (2)
              do j = 1, 2
                 if (entry(j)%is_valid) then
                    n_record = n_record + 1
                    allocate (new(n_record)%entry (2))
                    new(n_record)%entry(j) = entry(j)
                 end if
              end do
           end select
        end associate
     end do
  call move_alloc (from = new, to = beam_structure%record)
end subroutine beam_structure_expand
```

9.1.3   Polarization

To record polarization, we provide an allocatable array of `smatrix` objects,
sparse matrices. The polarization structure is independent of the structure-
function setup, they are combined only when an actual beam object is con-
structed.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: final_pol => beam_structure_final_pol
    procedure :: init_pol => beam_structure_init_pol
```

⟨*Beam structures: procedures*⟩+≡
```
    subroutine beam_structure_final_pol (beam_structure)
      class(beam_structure_t), intent(inout) :: beam_structure
      if (allocated (beam_structure%smatrix))  deallocate (beam_structure%smatrix)
      if (allocated (beam_structure%pol_f))  deallocate (beam_structure%pol_f)
    end subroutine beam_structure_final_pol

    subroutine beam_structure_init_pol (beam_structure, n)
      class(beam_structure_t), intent(inout) :: beam_structure
      integer, intent(in) :: n
      if (allocated (beam_structure%smatrix))  deallocate (beam_structure%smatrix)
      allocate (beam_structure%smatrix (n))
      if (.not. allocated (beam_structure%pol_f)) &
           allocate (beam_structure%pol_f (n), source = 1._default)
    end subroutine beam_structure_init_pol
```

Directly copy the spin density matrices.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: set_smatrix => beam_structure_set_smatrix
```

⟨Beam structures: procedures⟩+≡
```
    subroutine beam_structure_set_smatrix (beam_structure, i, smatrix)
      class(beam_structure_t), intent(inout) :: beam_structure
      integer, intent(in) :: i
      type(smatrix_t), intent(in) :: smatrix
      beam_structure%smatrix(i) = smatrix
    end subroutine beam_structure_set_smatrix
```

Initialize one of the spin density matrices manually.
⟨Beam structures: beam structure: TBP⟩+≡
```
    procedure :: init_smatrix => beam_structure_init_smatrix
```
⟨Beam structures: procedures⟩+≡
```
    subroutine beam_structure_init_smatrix (beam_structure, i, n_entry)
      class(beam_structure_t), intent(inout) :: beam_structure
      integer, intent(in) :: i
      integer, intent(in) :: n_entry
      call beam_structure%smatrix(i)%init (2, n_entry)
    end subroutine beam_structure_init_smatrix
```

Set a polarization entry.
⟨Beam structures: beam structure: TBP⟩+≡
```
    procedure :: set_sentry => beam_structure_set_sentry
```
⟨Beam structures: procedures⟩+≡
```
    subroutine beam_structure_set_sentry &
        (beam_structure, i, i_entry, index, value)
      class(beam_structure_t), intent(inout) :: beam_structure
      integer, intent(in) :: i
      integer, intent(in) :: i_entry
      integer, dimension(:), intent(in) :: index
      complex(default), intent(in) :: value
      call beam_structure%smatrix(i)%set_entry (i_entry, index, value)
    end subroutine beam_structure_set_sentry
```

Set the array of polarization fractions.
⟨Beam structures: beam structure: TBP⟩+≡
```
    procedure :: set_pol_f => beam_structure_set_pol_f
```
⟨Beam structures: procedures⟩+≡
```
    subroutine beam_structure_set_pol_f (beam_structure, f)
      class(beam_structure_t), intent(inout) :: beam_structure
      real(default), dimension(:), intent(in) :: f
      if (allocated (beam_structure%pol_f))  deallocate (beam_structure%pol_f)
      allocate (beam_structure%pol_f (size (f)), source = f)
    end subroutine beam_structure_set_pol_f
```

9.1.4   Beam momenta

By default, beam momenta are deduced from the `sqrts` value or from the mass of the decaying particle, assuming a c.m. setup. Here we set them explicitly.
⟨Beam structures: beam structure: TBP⟩+≡
```
    procedure :: final_mom => beam_structure_final_mom
```

⟨*Beam structures: procedures*⟩+≡
```
    subroutine beam_structure_final_mom (beam_structure)
      class(beam_structure_t), intent(inout) :: beam_structure
      if (allocated (beam_structure%p))  deallocate (beam_structure%p)
      if (allocated (beam_structure%theta))  deallocate (beam_structure%theta)
      if (allocated (beam_structure%phi))  deallocate (beam_structure%phi)
    end subroutine beam_structure_final_mom
```

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: set_momentum => beam_structure_set_momentum
    procedure :: set_theta => beam_structure_set_theta
    procedure :: set_phi => beam_structure_set_phi
```
⟨*Beam structures: procedures*⟩+≡
```
    subroutine beam_structure_set_momentum (beam_structure, p)
      class(beam_structure_t), intent(inout) :: beam_structure
      real(default), dimension(:), intent(in) :: p
      if (allocated (beam_structure%p))  deallocate (beam_structure%p)
      allocate (beam_structure%p (size (p)), source = p)
    end subroutine beam_structure_set_momentum

    subroutine beam_structure_set_theta (beam_structure, theta)
      class(beam_structure_t), intent(inout) :: beam_structure
      real(default), dimension(:), intent(in) :: theta
      if (allocated (beam_structure%theta))  deallocate (beam_structure%theta)
      allocate (beam_structure%theta (size (theta)), source = theta)
    end subroutine beam_structure_set_theta

    subroutine beam_structure_set_phi (beam_structure, phi)
      class(beam_structure_t), intent(inout) :: beam_structure
      real(default), dimension(:), intent(in) :: phi
      if (allocated (beam_structure%phi))  deallocate (beam_structure%phi)
      allocate (beam_structure%phi (size (phi)), source = phi)
    end subroutine beam_structure_set_phi
```

9.1.5 Get contents

Look at the incoming particles. We may also have the case that beam particles
are not specified, but polarization.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: is_set => beam_structure_is_set
    procedure :: get_n_beam => beam_structure_get_n_beam
    procedure :: get_prt => beam_structure_get_prt
```
⟨*Beam structures: procedures*⟩+≡
```
    function beam_structure_is_set (beam_structure) result (flag)
      class(beam_structure_t), intent(in) :: beam_structure
      logical :: flag
      flag = beam_structure%n_beam > 0 .or. beam_structure%asymmetric ()
    end function beam_structure_is_set

    function beam_structure_get_n_beam (beam_structure) result (n)
      class(beam_structure_t), intent(in) :: beam_structure
```

```
      integer :: n
      n = beam_structure%n_beam
    end function beam_structure_get_n_beam

    function beam_structure_get_prt (beam_structure) result (prt)
      class(beam_structure_t), intent(in) :: beam_structure
      type(string_t), dimension(:), allocatable :: prt
      allocate (prt (size (beam_structure%prt)))
      prt = beam_structure%prt
    end function beam_structure_get_prt
```

Return the number of records.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: get_n_record => beam_structure_get_n_record
```

⟨*Beam structures: procedures*⟩+≡
```
    function beam_structure_get_n_record (beam_structure) result (n)
      class(beam_structure_t), intent(in) :: beam_structure
      integer :: n
      if (allocated (beam_structure%record)) then
         n = size (beam_structure%record)
      else
         n = 0
      end if
    end function beam_structure_get_n_record
```

Return an array consisting of the beam indices affected by the valid entries within a record. After expansion, there should be exactly one valid entry per record.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: get_i_entry => beam_structure_get_i_entry
```

⟨*Beam structures: procedures*⟩+≡
```
    function beam_structure_get_i_entry (beam_structure, i) result (i_entry)
      class(beam_structure_t), intent(in) :: beam_structure
      integer, intent(in) :: i
      integer, dimension(:), allocatable :: i_entry
      associate (record => beam_structure%record(i))
        select case (size (record%entry))
        case (1)
           if (record%entry(1)%is_valid) then
              allocate (i_entry (2), source = [1, 2])
           else
              allocate (i_entry (0))
           end if
        case (2)
           if (all (record%entry%is_valid)) then
              allocate (i_entry (2), source = [1, 2])
           else if (record%entry(1)%is_valid) then
              allocate (i_entry (1), source = [1])
           else if (record%entry(2)%is_valid) then
              allocate (i_entry (1), source = [2])
           else
              allocate (i_entry (0))
```

```
          end if
        end select
      end associate
    end function beam_structure_get_i_entry
```

Return the name of the first valid entry within a record. After expansion, there should be exactly one valid entry per record.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: get_name => beam_structure_get_name
```

⟨*Beam structures: procedures*⟩+≡
```
    function beam_structure_get_name (beam_structure, i) result (name)
      class(beam_structure_t), intent(in) :: beam_structure
      integer, intent(in) :: i
      type(string_t) :: name
      associate (record => beam_structure%record(i))
        if (record%entry(1)%is_valid) then
           name = record%entry(1)%name
        else if (size (record%entry) == 2) then
           name = record%entry(2)%name
        end if
      end associate
    end function beam_structure_get_name
```

Return true if the beam structure contains a particular structure function identifier (such as lhapdf, isr, etc.)

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: contains => beam_structure_contains
```

⟨*Beam structures: procedures*⟩+≡
```
    function beam_structure_contains (beam_structure, name) result (flag)
      class(beam_structure_t), intent(in) :: beam_structure
      character(*), intent(in) :: name
      logical :: flag
      integer :: i, j
      flag = .false.
      if (allocated (beam_structure%record)) then
         do i = 1, size (beam_structure%record)
            do j = 1, size (beam_structure%record(i)%entry)
               flag = beam_structure%record(i)%entry(j)%name == name
               if (flag)  return
            end do
         end do
      end if
    end function beam_structure_contains
```

Return polarization data.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
    procedure :: polarized => beam_structure_polarized
    procedure :: get_smatrix => beam_structure_get_smatrix
    procedure :: get_pol_f => beam_structure_get_pol_f
    procedure :: asymmetric => beam_structure_asymmetric
```

⟨*Beam structures: procedures*⟩+≡
```
function beam_structure_polarized (beam_structure) result (flag)
  class(beam_structure_t), intent(in) :: beam_structure
  logical :: flag
  flag = allocated (beam_structure%smatrix)
end function beam_structure_polarized

function beam_structure_get_smatrix (beam_structure) result (smatrix)
  class(beam_structure_t), intent(in) :: beam_structure
  type(smatrix_t), dimension(:), allocatable :: smatrix
  allocate (smatrix (size (beam_structure%smatrix)), &
       source = beam_structure%smatrix)
end function beam_structure_get_smatrix

function beam_structure_get_pol_f (beam_structure) result (pol_f)
  class(beam_structure_t), intent(in) :: beam_structure
  real(default), dimension(:), allocatable :: pol_f
  allocate (pol_f (size (beam_structure%pol_f)), &
       source = beam_structure%pol_f)
end function beam_structure_get_pol_f

function beam_structure_asymmetric (beam_structure) result (flag)
  class(beam_structure_t), intent(in) :: beam_structure
  logical :: flag
  flag = allocated (beam_structure%p) &
       .or. allocated (beam_structure%theta) &
       .or. allocated (beam_structure%phi)
end function beam_structure_asymmetric
```

Return the beam momenta (the space part, i.e., three-momenta). This is meaningful only if momenta and, optionally, angles have been set.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
procedure :: get_momenta => beam_structure_get_momenta
```

⟨*Beam structures: procedures*⟩+≡
```
function beam_structure_get_momenta (beam_structure) result (p)
  class(beam_structure_t), intent(in) :: beam_structure
  type(vector3_t), dimension(:), allocatable :: p
  real(default), dimension(:), allocatable :: theta, phi
  integer :: n, i
  if (allocated (beam_structure%p)) then
     n = size (beam_structure%p)
     if (allocated (beam_structure%theta)) then
        if (size (beam_structure%theta) == n) then
           allocate (theta (n), source = beam_structure%theta)
        else
           call msg_fatal ("Beam structure: mismatch in momentum vs. &
                &angle theta specification")
        end if
     else
        allocate (theta (n), source = 0._default)
     end if
     if (allocated (beam_structure%phi)) then
        if (size (beam_structure%phi) == n) then
```

957

```
            allocate (phi (n), source = beam_structure%phi)
         else
            call msg_fatal ("Beam structure: mismatch in momentum vs. &
                  &angle phi specification")
         end if
      else
         allocate (phi (n), source = 0._default)
      end if
      allocate (p (n))
      do i = 1, n
         p(i) = beam_structure%p(i) * vector3_moving ([ &
               sin (theta(i)) * cos (phi(i)), &
               sin (theta(i)) * sin (phi(i)), &
               cos (theta(i))])
      end do
      if (n == 2)  p(2) = - p(2)
   else
      call msg_fatal ("Beam structure: angle theta/phi specified but &
            &momentum/a p undefined")
   end if
end function beam_structure_get_momenta
```

9.1.6 Unit Tests

⟨*Beam structures: public*⟩+≡
```
    public :: beam_structures_test
```

⟨*Beam structures: tests*⟩≡
```
    subroutine beam_structures_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Beam structures: execute tests*⟩
```
  end subroutine beam_structures_test
```

Empty structure

⟨*Beam structures: execute tests*⟩≡
```
    call test (beam_structures_1, "beam_structures_1", &
         "empty beam structure record", &
         u, results)
```

⟨*Beam structures: tests*⟩+≡
```
    subroutine beam_structures_1 (u)
      integer, intent(in) :: u
      type(beam_structure_t) :: beam_structure

      write (u, "(A)")  "* Test output: beam_structures_1"
      write (u, "(A)")  "*   Purpose: display empty beam structure record"
      write (u, "(A)")

      call beam_structure%write (u)
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Test output end: beam_structures_1"

   end subroutine beam_structures_1
```

Nontrivial configurations

⟨*Beam structures: execute tests*⟩+≡
```
      call test (beam_structures_2, "beam_structures_2", &
            "beam structure records", &
            u, results)
```
⟨*Beam structures: tests*⟩+≡
```
   subroutine beam_structures_2 (u)
     integer, intent(in) :: u
     type(beam_structure_t) :: beam_structure
     integer, dimension(0) :: empty_array
     type(string_t) :: s

     write (u, "(A)")   "* Test output: beam_structures_2"
     write (u, "(A)")   "*   Purpose: setup beam structure records"
     write (u, "(A)")

     s = "s"

     call beam_structure%init_sf ([s], empty_array)
     call beam_structure%write (u)

     write (u, "(A)")

     call beam_structure%init_sf ([s, s], [1])
     call beam_structure%set_sf (1, 1, var_str ("a"))
     call beam_structure%write (u)

     write (u, "(A)")

     call beam_structure%init_sf ([s, s], [2])
     call beam_structure%set_sf (1, 1, var_str ("a"))
     call beam_structure%set_sf (1, 2, var_str ("b"))
     call beam_structure%write (u)

     write (u, "(A)")

     call beam_structure%init_sf ([s, s], [2, 1])
     call beam_structure%set_sf (1, 1, var_str ("a"))
     call beam_structure%set_sf (1, 2, var_str ("b"))
     call beam_structure%set_sf (2, 1, var_str ("c"))
     call beam_structure%write (u)

     write (u, "(A)")
     write (u, "(A)")   "* Test output end: beam_structures_2"

   end subroutine beam_structures_2
```

Expansion

Provide a function that tells, for the dummy structure function names used here, whether they are considered a two-particle spectrum or a single-particle structure function:

⟨*Beam structures: tests*⟩+≡
```
    function test_strfun_mode (name) result (n)
      type(string_t), intent(in) :: name
      integer :: n
      select case (char (name))
      case ("a");  n = 2
      case ("b");  n = 1
      case default;  n = 0
      end select
    end function test_strfun_mode
```

⟨*Beam structures: execute tests*⟩+≡
```
    call test (beam_structures_3, "beam_structures_3", &
        "beam structure expansion", &
        u, results)
```

⟨*Beam structures: tests*⟩+≡
```
    subroutine beam_structures_3 (u)
      integer, intent(in) :: u
      type(beam_structure_t) :: beam_structure
      type(string_t) :: s

      write (u, "(A)")  "* Test output: beam_structures_3"
      write (u, "(A)")  "*   Purpose: expand beam structure records"
      write (u, "(A)")

      s = "s"

      write (u, "(A)")  "* Pair spectrum (keep as-is)"
      write (u, "(A)")

      call beam_structure%init_sf ([s, s], [1])
      call beam_structure%set_sf (1, 1, var_str ("a"))
      call beam_structure%write (u)

      write (u, "(A)")

      call beam_structure%expand (test_strfun_mode)
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Structure function pair (expand)"
      write (u, "(A)")

      call beam_structure%init_sf ([s, s], [2])
      call beam_structure%set_sf (1, 1, var_str ("b"))
      call beam_structure%set_sf (1, 2, var_str ("b"))
      call beam_structure%write (u)
```

```
      write (u, "(A)")

      call beam_structure%expand (test_strfun_mode)
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Structure function (separate and expand)"
      write (u, "(A)")

      call beam_structure%init_sf ([s, s], [1])
      call beam_structure%set_sf (1, 1, var_str ("b"))
      call beam_structure%write (u)

      write (u, "(A)")

      call beam_structure%expand (test_strfun_mode)
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Combination"
      write (u, "(A)")

      call beam_structure%init_sf ([s, s], [1, 1])
      call beam_structure%set_sf (1, 1, var_str ("a"))
      call beam_structure%set_sf (2, 1, var_str ("b"))
      call beam_structure%write (u)

      write (u, "(A)")

      call beam_structure%expand (test_strfun_mode)
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: beam_structures_3"

    end subroutine beam_structures_3
```

Public methods

Check the methods that can be called to get the beam-structure contents.

⟨*Beam structures: execute tests*⟩+≡
```
    call test (beam_structures_4, "beam_structures_4", &
        "beam structure contents", &
        u, results)
```

⟨*Beam structures: tests*⟩+≡
```
    subroutine beam_structures_4 (u)
      integer, intent(in) :: u
      type(beam_structure_t) :: beam_structure
      type(string_t) :: s
      type(string_t), dimension(2) :: prt
      integer :: i
```

```
write (u, "(A)")  "* Test output: beam_structures_4"
write (u, "(A)")  "*   Purpose: check the API"
write (u, "(A)")

s = "s"

write (u, "(A)")  "* Structure-function combination"
write (u, "(A)")

call beam_structure%init_sf ([s, s], [1, 2, 2])
call beam_structure%set_sf (1, 1, var_str ("a"))
call beam_structure%set_sf (2, 1, var_str ("b"))
call beam_structure%set_sf (3, 2, var_str ("c"))
call beam_structure%write (u)

write (u, *)
write (u, "(1x,A,I0)")  "n_beam = ", beam_structure%get_n_beam ()
prt = beam_structure%get_prt ()
write (u, "(1x,A,2(1x,A))")  "prt =", char (prt(1)), char (prt(2))

write (u, *)
write (u, "(1x,A,I0)")  "n_record = ", beam_structure%get_n_record ()

do i = 1, 3
   write (u, "(A)")
   write (u, "(1x,A,I0,A,A)")  "name(", i, ") = ", &
        char (beam_structure%get_name (i))
   write (u, "(1x,A,I0,A,2(1x,I0))")  "i_entry(", i, ") =", &
        beam_structure%get_i_entry (i)
end do

write (u, "(A)")
write (u, "(A)")  "* Test output end: beam_structures_4"

end subroutine beam_structures_4
```

Polarization

The polarization properties are independent from the structure-function setup.

⟨*Beam structures: execute tests*⟩+≡
```
call test (beam_structures_5, "beam_structures_5", &
     "polarization", &
     u, results)
```

⟨*Beam structures: tests*⟩+≡
```
subroutine beam_structures_5 (u)
  integer, intent(in) :: u
  type(beam_structure_t) :: beam_structure
  integer, dimension(0) :: empty_array
  type(string_t) :: s

  write (u, "(A)")  "* Test output: beam_structures_5"
  write (u, "(A)")  "*   Purpose: setup polarization in beam structure records"
```

```
      write (u, "(A)")

      s = "s"

      call beam_structure%init_sf ([s], empty_array)
      call beam_structure%init_pol (1)
      call beam_structure%init_smatrix (1, 1)
      call beam_structure%set_sentry (1, 1, [0,0], (1._default, 0._default))
      call beam_structure%set_pol_f ([0.5_default])
      call beam_structure%write (u)


      write (u, "(A)")
      call beam_structure%final_sf ()
      call beam_structure%final_pol ()

      call beam_structure%init_sf ([s, s], [1])
      call beam_structure%set_sf (1, 1, var_str ("a"))
      call beam_structure%init_pol (2)
      call beam_structure%init_smatrix (1, 2)
      call beam_structure%set_sentry (1, 1, [-1,1], (0.5_default,-0.5_default))
      call beam_structure%set_sentry (1, 2, [ 1,1], (1._default, 0._default))
      call beam_structure%init_smatrix (2, 0)
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: beam_structures_5"

    end subroutine beam_structures_5
```

Momenta

The momenta are independent from the structure-function setup.

⟨*Beam structures: execute tests*⟩+≡
```
    call test (beam_structures_6, "beam_structures_6", &
         "momenta", &
         u, results)
```

⟨*Beam structures: tests*⟩+≡
```
    subroutine beam_structures_6 (u)
      integer, intent(in) :: u
      type(beam_structure_t) :: beam_structure
      integer, dimension(0) :: empty_array
      type(string_t) :: s

      write (u, "(A)")  "* Test output: beam_structures_6"
      write (u, "(A)")  "*   Purpose: setup momenta in beam structure records"
      write (u, "(A)")

      s = "s"

      call beam_structure%init_sf ([s], empty_array)
      call beam_structure%set_momentum ([500._default])
```

```
      call beam_structure%write (u)


      write (u, "(A)")
      call beam_structure%final_sf ()
      call beam_structure%final_mom ()

      call beam_structure%init_sf ([s, s], [1])
      call beam_structure%set_momentum ([500._default, 700._default])
      call beam_structure%set_theta ([0._default, 0.1_default])
      call beam_structure%set_phi ([0._default, 1.51_default])
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: beam_structures_6"

    end subroutine beam_structures_6
```

9.2   Beams for collisions and decays

⟨beams.f90⟩≡
  ⟨*File header*⟩

  module beams

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants !NODEP!
  ⟨*Use file utils*⟩
    use limits, only: FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use md5
    use models
    use flavors
    use colors
    use polarizations
    use quantum_numbers
    use state_matrices
    use interactions
    use beam_structures

  ⟨*Standard module head*⟩

  ⟨*Beams: public*⟩

  ⟨*Beams: types*⟩

  ⟨*Beams: interfaces*⟩

  contains
```

⟨*Beams: procedures*⟩

⟨*Beams: tests*⟩

```
end module beams
```

9.2.1 Beam data

The beam data type contains beam data for one or two beams, depending on whether we are dealing with beam collisions or particle decay. In addition, it holds the c.m. energy `sqrts`, the Lorentz transformation `L` that transforms the c.m. system into the lab system, and the pair of c.m. momenta.

⟨*Beams: public*⟩≡
```
    public :: beam_data_t
```

⟨*Beams: types*⟩≡
```
    type :: beam_data_t
       logical :: initialized = .false.
       integer :: n = 0
       type(flavor_t), dimension(:), allocatable :: flv
       real(default), dimension(:), allocatable :: mass
       type(pmatrix_t), dimension(:), allocatable :: pmatrix
       logical :: lab_is_cm_frame = .true.
       type(vector4_t), dimension(:), allocatable :: p_cm
       type(vector4_t), dimension(:), allocatable :: p
       type(lorentz_transformation_t), allocatable  :: L_cm_to_lab
       real(default) :: sqrts = 0
       character(32) :: md5sum = ""
     contains
     ⟨Beams: beam data: TBP⟩
    end type beam_data_t
```

Generic initializer. This is called by the specific initializers below. Initialize either for decay or for collision.

⟨*Beams: procedures*⟩≡
```
    subroutine beam_data_init (beam_data, n)
      type(beam_data_t), intent(out) :: beam_data
      integer, intent(in) :: n
      beam_data%n = n
      allocate (beam_data%flv (n))
      allocate (beam_data%mass (n))
      allocate (beam_data%pmatrix (n))
      allocate (beam_data%p_cm (n))
      allocate (beam_data%p (n))
      beam_data%initialized = .true.
    end subroutine beam_data_init
```

Finalizer: needed for the polarization components of the beams.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_final
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_final (beam_data)
      type(beam_data_t), intent(inout) :: beam_data
      beam_data%initialized = .false.
    end subroutine beam_data_final
```

The verbose (default) version is for debugging. The short version is for screen
output in the UI.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_write
```

⟨*Beams: beam data: TBP*⟩≡
```
    procedure :: write => beam_data_write
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_write (beam_data, unit, verbose, write_md5sum)
      class(beam_data_t), intent(in) :: beam_data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, write_md5sum
      integer :: prt_name_len
      logical :: verb, write_md5
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      verb = .false.;  if (present (verbose))  verb = verbose
      write_md5 = verb;  if (present (write_md5sum)) write_md5 = write_md5sum
      if (.not. beam_data%initialized) then
         write (u, "(1x,A)") "Beam data: [undefined]"
         return
      end if
      prt_name_len = maxval (len (flavor_get_name (beam_data%flv)))
      select case (beam_data%n)
      case (1)
         write (u, "(1x,A)") "Beam data (decay):"
         if (verb) then
            call write_prt (1)
            call beam_data%pmatrix(1)%write (u)
            write (u, *) "R.f. momentum:"
            call vector4_write (beam_data%p_cm(1), u)
            write (u, *) "Lab momentum:"
            call vector4_write (beam_data%p(1), u)
         else
            call write_prt (1)
         end if
      case (2)
         write (u, "(1x,A)") "Beam data (collision):"
         if (verb) then
            call write_prt (1)
            call beam_data%pmatrix(1)%write (u)
            call write_prt (2)
            call beam_data%pmatrix(2)%write (u)
            call write_sqrts
            write (u, *) "C.m. momenta:"
            call vector4_write (beam_data%p_cm(1), u)
            call vector4_write (beam_data%p_cm(2), u)
            write (u, *) "Lab momenta:"
```

```
          call vector4_write (beam_data%p(1), u)
          call vector4_write (beam_data%p(2), u)
      else
          call write_prt (1)
          call write_prt (2)
          call write_sqrts
      end if
    end select
    if (allocated (beam_data%L_cm_to_lab)) then
      if (verb) then
          call lorentz_transformation_write (beam_data%L_cm_to_lab, u)
      else
          write (u, "(1x,A)")  "Beam structure: lab and c.m. frame differ"
      end if
    end if
    if (write_md5) then
      write (u, *) "MD5 sum: ", beam_data%md5sum
    end if
  contains
    subroutine write_sqrts
      character(80) :: sqrts_str
      write (sqrts_str, "(" // FMT_19 // ")")  beam_data%sqrts
      write (u, "(3x,A)")  "sqrts = " // trim (adjustl (sqrts_str)) // " GeV"
    end subroutine write_sqrts
    subroutine write_prt (i)
      integer, intent(in) :: i
      character(80) :: name_str, mass_str
      write (name_str, "(A)")  char (flavor_get_name (beam_data%flv(i)))
      write (mass_str, "(ES13.7)")  beam_data%mass(i)
      write (u, "(3x,A)", advance="no") &
          name_str(:prt_name_len) // "  (mass = " &
          // trim (adjustl (mass_str)) // " GeV)"
      if (beam_data%pmatrix(i)%is_polarized ()) then
        write (u, "(2x,A)")  "polarized"
      else
        write (u, *)
      end if
    end subroutine write_prt
  end subroutine beam_data_write
```

Return initialization status:

⟨*Beams: public*⟩+≡
```
    public :: beam_data_are_valid
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_are_valid (beam_data) result (flag)
      logical :: flag
      type(beam_data_t), intent(in) :: beam_data
      flag = beam_data%initialized
    end function beam_data_are_valid
```

Check whether beam data agree with the current values of relevant parameters.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_check_scattering
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_check_scattering (beam_data, sqrts)
      type(beam_data_t), intent(in) :: beam_data
      real(default), intent(in), optional :: sqrts
      if (beam_data_are_valid (beam_data)) then
          if (present (sqrts)) then
              if (sqrts /= beam_data%sqrts) then
                  call msg_error ("Current setting of sqrts is inconsistent " &
                        // "with beam setup (ignored).")
              end if
          end if
      else
          call msg_bug ("Beam setup: invalid beam data")
      end if
    end subroutine beam_data_check_scattering
```

Return the number of beams (1 for decays, 2 for collisions).
⟨*Beams: public*⟩+≡
```
    public :: beam_data_get_n_in
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_get_n_in (beam_data) result (n_in)
      integer :: n_in
      type(beam_data_t), intent(in) :: beam_data
      n_in = beam_data%n
    end function beam_data_get_n_in
```

Return the beam flavor
⟨*Beams: public*⟩+≡
```
    public :: beam_data_get_flavor
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_get_flavor (beam_data) result (flv)
      type(flavor_t), dimension(:), allocatable :: flv
      type(beam_data_t), intent(in) :: beam_data
      allocate (flv (beam_data%n))
      flv = beam_data%flv
    end function beam_data_get_flavor
```

Return the beam energies
⟨*Beams: public*⟩+≡
```
    public :: beam_data_get_energy
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_get_energy (beam_data) result (e)
      real(default), dimension(:), allocatable :: e
      type(beam_data_t), intent(in) :: beam_data
      allocate (e (beam_data%n))
      if (beam_data%initialized) then
          e = energy (beam_data%p)
      else
          e = 0
      end if
    end function beam_data_get_energy
```

Return the c.m. energy.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_get_sqrts
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_get_sqrts (beam_data) result (sqrts)
      real(default) :: sqrts
      type(beam_data_t), intent(in) :: beam_data
      sqrts = beam_data%sqrts
    end function beam_data_get_sqrts
```

Return true if the lab and c.m. frame are specified as identical.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_cm_frame
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_cm_frame (beam_data) result (flag)
      type(beam_data_t), intent(in) :: beam_data
      logical :: flag
      flag = beam_data%lab_is_cm_frame
    end function beam_data_cm_frame
```

Return a MD5 checksum for beam data. If no checksum is present (because beams have not been initialized), compute the checksum of the sqrts value.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_get_md5sum
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_get_md5sum (beam_data, sqrts) result (md5sum_beams)
      type(beam_data_t), intent(in) :: beam_data
      real(default), intent(in) :: sqrts
      character(32) :: md5sum_beams
      character(80) :: buffer
      if (beam_data%md5sum /= "") then
         md5sum_beams = beam_data%md5sum
      else
         write (buffer, *)  sqrts
         md5sum_beams = md5sum (buffer)
      end if
    end function beam_data_get_md5sum
```

9.2.2 Initializers: beam structure

Initialize the beam data object from a beam structure object, given energy and model.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_init_structure
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_init_structure &
        (beam_data, structure, sqrts, model, decay_rest_frame)
      type(beam_data_t), intent(out) :: beam_data
```

```fortran
      type(beam_structure_t), intent(in) :: structure
      integer :: n_beam
      real(default), intent(in) :: sqrts
      type(model_t), intent(in), target :: model
      logical, intent(in), optional :: decay_rest_frame
      type(flavor_t), dimension(:), allocatable :: flv
      n_beam = structure%get_n_beam ()
      allocate (flv (n_beam))
      call flavor_init (flv, structure%get_prt (), model)
      if (structure%asymmetric ()) then
         if (structure%polarized ()) then
            call beam_data_init_momenta (beam_data, &
                 structure%get_momenta (), flv, &
                 structure%get_smatrix (), structure%get_pol_f ())
         else
            call beam_data_init_momenta (beam_data, &
                 structure%get_momenta (), flv)
         end if
      else
         select case (n_beam)
         case (1)
            if (structure%polarized ()) then
               call beam_data_init_decay (beam_data, flv, &
                    structure%get_smatrix (), structure%get_pol_f (), &
                    rest_frame = decay_rest_frame)
            else
               call beam_data_init_decay (beam_data, flv, &
                    rest_frame = decay_rest_frame)
            end if
         case (2)
            if (structure%polarized ()) then
               call beam_data_init_sqrts (beam_data, sqrts, flv, &
                    structure%get_smatrix (), structure%get_pol_f ())
            else
               call beam_data_init_sqrts (beam_data, sqrts, flv)
            end if
         case default
            call msg_bug ("Beam data: invalid beam structure object")
         end select
      end if
    end subroutine beam_data_init_structure
```

9.2.3   Initializers: collisions

This is the simplest one: just the two flavors, c.m. energy, polarization. Color
is inferred from flavor. Beam momenta and c.m. momenta coincide.

⟨*Beams: public*⟩+≡
```fortran
    public :: beam_data_init_sqrts
```

⟨*Beams: procedures*⟩+≡
```fortran
    subroutine beam_data_init_sqrts (beam_data, sqrts, flv, smatrix, pol_f)
      type(beam_data_t), intent(out) :: beam_data
      real(default), intent(in) :: sqrts
```

```
type(flavor_t), dimension(:), intent(in) :: flv
type(smatrix_t), dimension(:), intent(in), optional :: smatrix
real(default), dimension(:), intent(in), optional :: pol_f
real(default), dimension(size(flv)) :: E, p
call beam_data_init (beam_data, size (flv))
beam_data%sqrts = sqrts
beam_data%lab_is_cm_frame = .true.
select case (beam_data%n)
case (1)
   E = sqrts;  p = 0
   beam_data%p_cm = vector4_moving (E, p, 3)
   beam_data%p = beam_data%p_cm
case (2)
   beam_data%p_cm = colliding_momenta (sqrts, flavor_get_mass (flv))
   beam_data%p = colliding_momenta (sqrts, flavor_get_mass (flv))
end select
call beam_data_finish_initialization (beam_data, flv, smatrix, pol_f)
end subroutine beam_data_init_sqrts
```

This version sets beam momenta directly, assuming that they are asymmetric,
i.e., lab frame and c.m. frame do not coincide. Polarization info is deferred to a
common initializer.

The Lorentz transformation that we compute here is not actually used in
the calculation; instead, it will be recomputed for each event in the subroutine
`phs_set_incoming_momenta`. We compute it here for the nominal beam setup
nevertheless, so we can print it and, in particular, include it in the MD5 sum.

⟨*Beams: public*⟩+≡
```
public :: beam_data_init_momenta
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_data_init_momenta (beam_data, p3, flv, smatrix, pol_f)
  type(beam_data_t), intent(out) :: beam_data
  type(vector3_t), dimension(:), intent(in) :: p3
  type(flavor_t), dimension(:), intent(in) :: flv
  type(smatrix_t), dimension(:), intent(in), optional :: smatrix
  real(default), dimension(:), intent(in), optional :: pol_f
  type(vector4_t) :: p0
  type(vector4_t), dimension(:), allocatable :: p, p_cm_rot
  real(default), dimension(size(p3)) :: e
  real(default), dimension(size(flv)) :: m
  type(lorentz_transformation_t) :: L_boost, L_rot
  call beam_data_init (beam_data, size (flv))
  m = flavor_get_mass (flv)
  e = sqrt (p3 ** 2 + m ** 2)
  allocate (p (beam_data%n))
  p = vector4_moving (e, p3)
  p0 = sum (p)
  beam_data%p = p
  beam_data%lab_is_cm_frame = .false.
  beam_data%sqrts = p0 ** 1
  L_boost = boost (p0, beam_data%sqrts)
  allocate (p_cm_rot (beam_data%n))
  p_cm_rot = inverse (L_boost) * p
```

971

```
        allocate (beam_data%L_cm_to_lab)
        select case (beam_data%n)
        case (1)
           beam_data%L_cm_to_lab = L_boost
           beam_data%p_cm = vector4_at_rest (beam_data%sqrts)
        case (2)
           L_rot = rotation_to_2nd (3, space_part (p_cm_rot(1)))
           beam_data%L_cm_to_lab = L_boost * L_rot
           beam_data%p_cm = &
                colliding_momenta (beam_data%sqrts, flavor_get_mass (flv))
        end select
        call beam_data_finish_initialization (beam_data, flv, smatrix, pol_f)
      end subroutine beam_data_init_momenta
```

Final steps: If requested, rotate the beams in the lab frame, and set the beam-data components.

⟨*Beams: procedures*⟩+≡

```
      subroutine beam_data_finish_initialization (beam_data, flv, smatrix, pol_f)
        type(beam_data_t), intent(inout) :: beam_data
        type(flavor_t), dimension(:), intent(in) :: flv
        type(smatrix_t), dimension(:), intent(in), optional :: smatrix
        real(default), dimension(:), intent(in), optional :: pol_f
        integer :: i
        do i = 1, beam_data%n
           beam_data%flv(i) = flv(i)
           beam_data%mass(i) = flavor_get_mass (flv(i))
           if (present (smatrix)) then
              if (size (smatrix) /= beam_data%n) &
                   call msg_fatal ("Beam data: &
                   &polarization density array has wrong dimension")
              beam_data%pmatrix(i) = smatrix(i)
              if (present (pol_f)) then
                 if (size (pol_f) /= size (smatrix)) &
                      call msg_fatal ("Beam data: &
                      &polarization fraction array has wrong dimension")
                 call beam_data%pmatrix(i)%normalize (flv(i), pol_f(i))
              else
                 call beam_data%pmatrix(i)%normalize (flv(i), 1._default)
              end if
           else
              call beam_data%pmatrix(i)%init (2, 0)
              call beam_data%pmatrix(i)%normalize (flv(i), 0._default)
           end if
        end do
        call beam_data_compute_md5sum (beam_data)
      end subroutine beam_data_finish_initialization
```

The MD5 sum is stored within the beam-data record, so it can be checked for integrity in subsequent runs.

⟨*Beams: procedures*⟩+≡

```
      subroutine beam_data_compute_md5sum (beam_data)
        type(beam_data_t), intent(inout) :: beam_data
        integer :: unit
```

```
    unit = free_unit ()
    open (unit = unit, status = "scratch", action = "readwrite")
    call beam_data_write (beam_data, unit, write_md5sum = .false., &
        verbose = .true.)
    rewind (unit)
    beam_data%md5sum = md5sum (unit)
    close (unit)
  end subroutine beam_data_compute_md5sum
```

9.2.4 Initializers: decays

This is the simplest one: decay in rest frame. We need just flavor and polarization. Color is inferred from flavor. Beam momentum and c.m. momentum coincide.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_init_decay
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_data_init_decay (beam_data, flv, smatrix, pol_f, rest_frame)
      type(beam_data_t), intent(out) :: beam_data
      type(flavor_t), dimension(1), intent(in) :: flv
      type(smatrix_t), dimension(1), intent(in), optional :: smatrix
      real(default), dimension(:), intent(in), optional :: pol_f
      logical, intent(in), optional :: rest_frame
      real(default), dimension(1) :: m
      m = flavor_get_mass (flv)
      if (present (smatrix)) then
         call beam_data_init_sqrts (beam_data, m(1), flv, smatrix, pol_f)
      else
         call beam_data_init_sqrts (beam_data, m(1), flv, smatrix, pol_f)
      end if
      if (present (rest_frame))  beam_data%lab_is_cm_frame = rest_frame
    end subroutine beam_data_init_decay
```

9.2.5 Sanity check

After the beams have been set, the initial-particle masses may have been modified. This can be checked here.

⟨*Beams: public*⟩+≡
```
    public :: beam_data_masses_are_consistent
```

⟨*Beams: procedures*⟩+≡
```
    function beam_data_masses_are_consistent (beam_data) result (flag)
      logical :: flag
      type(beam_data_t), intent(in) :: beam_data
      flag = all (beam_data%mass == flavor_get_mass (beam_data%flv))
    end function beam_data_masses_are_consistent
```

9.2.6 The beams type

Beam objects are interaction objects that contain the actual beam data including polarization and density matrix. For collisions, the beam object actually contains two beams.

⟨*Beams: public*⟩+≡
```
    public :: beam_t
```

⟨*Beams: types*⟩+≡
```
    type :: beam_t
       private
       type(interaction_t) :: int
    end type beam_t
```

The constructor contains code that converts beam data into the (entangled) particle-pair quantum state. First, we set the number of particles and polarization mask. (The polarization mask is handed over to all later interactions, so if helicity is diagonal or absent, this fact is used when constructing the hard-interaction events.) Then, we construct the entangled state that combines helicity, flavor and color of the two particles (where flavor and color are unique, while several helicity states are possible). Then, we transfer this state together with the associated values from the spin density matrix into the `interaction_t` object.

⟨*Beams: public*⟩+≡
```
    public :: beam_init
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_init (beam, beam_data)
      type(beam_t), intent(out) :: beam
      type(beam_data_t), intent(in), target :: beam_data
      type(quantum_numbers_mask_t), dimension(beam_data%n) :: mask
      type(state_matrix_t), target :: state_hel, state_fc, state_tmp
      type(state_iterator_t) :: it_hel, it_tmp
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      type(polarization_t), dimension(:), allocatable :: pol
      integer :: i
      mask = new_quantum_numbers_mask (.false., .false., &
           .not. beam_data%pmatrix%is_polarized (), &
           mask_hd = beam_data%pmatrix%is_diagonal ())
      call interaction_init &
           (beam%int, 0, 0, beam_data%n, mask=mask, store_values=.true.)
      allocate (pol (beam_data%n))
      do i = 1, size (pol)
         call polarization_init_pmatrix (pol(i), beam_data%pmatrix(i))
      end do
      call combine_polarization_states (pol, state_hel)
      do i = 1, size (pol)
         call polarization_final (pol(i))
      end do
      allocate (qn (beam_data%n))
      call quantum_numbers_init &
           (qn, beam_data%flv, color_from_flavor (beam_data%flv, 1))
      call state_matrix_init (state_fc)
      call state_matrix_add_state (state_fc, qn)
```

```
      call merge_state_matrices (state_hel, state_fc, state_tmp)
      call state_iterator_init (it_hel, state_hel)
      call state_iterator_init (it_tmp, state_tmp)
      do while (state_iterator_is_valid (it_hel))
         call interaction_add_state (beam%int, &
              state_iterator_get_quantum_numbers (it_tmp), &
              value=state_iterator_get_matrix_element (it_hel))
         call state_iterator_advance (it_hel)
         call state_iterator_advance (it_tmp)
      end do
      call interaction_freeze (beam%int)
      call interaction_set_momenta &
           (beam%int, beam_data%p, outgoing = .true.)
      call state_matrix_final (state_hel)
      call state_matrix_final (state_fc)
      call state_matrix_final (state_tmp)
    end subroutine beam_init
```

Finalizer:

⟨*Beams: public*⟩+≡
```
    public :: beam_final
```

⟨*Beams: procedures*⟩+≡
```
    elemental subroutine beam_final (beam)
      type(beam_t), intent(inout) :: beam
      call interaction_final (beam%int)
    end subroutine beam_final
```

I/O:

⟨*Beams: public*⟩+≡
```
    public :: beam_write
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_write (beam, unit, verbose, show_momentum_sum, show_mass)
      type(beam_t), intent(in) :: beam
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (interaction_get_n_out (beam%int))
      case (1);  write (u, *) "Decaying particle:"
      case (2);  write (u, *) "Colliding beams:"
      end select
      call interaction_write &
           (beam%int, unit, verbose = verbose, show_momentum_sum = &
              show_momentum_sum, show_mass = show_mass)
    end subroutine beam_write
```

Defined assignment: deep copy

⟨*Beams: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Beams: interfaces*⟩≡
```
interface assignment(=)
   module procedure beam_assign
end interface
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_assign (beam_out, beam_in)
  type(beam_t), intent(out) :: beam_out
  type(beam_t), intent(in) :: beam_in
  beam_out%int = beam_in%int
end subroutine beam_assign
```

9.2.7 Inherited procedures

⟨*Beams: public*⟩+≡
```
public :: interaction_set_source_link
```

⟨*Beams: interfaces*⟩+≡
```
interface interaction_set_source_link
   module procedure interaction_set_source_link_beam
end interface
```

⟨*Beams: procedures*⟩+≡
```
subroutine interaction_set_source_link_beam (int, i, beam1, i1)
  type(interaction_t), intent(inout) :: int
  type(beam_t), intent(in), target :: beam1
  integer, intent(in) :: i, i1
  call interaction_set_source_link (int, i, beam1%int, i1)
end subroutine interaction_set_source_link_beam
```

9.2.8 Accessing contents

Return the interaction component – as a pointer, to avoid any copying.

⟨*Beams: public*⟩+≡
```
public :: beam_get_int_ptr
```

⟨*Beams: procedures*⟩+≡
```
function beam_get_int_ptr (beam) result (int)
  type(interaction_t), pointer :: int
  type(beam_t), intent(in), target :: beam
  int => beam%int
end function beam_get_int_ptr
```

Set beam momenta directly. (Used for cascade decays.)

⟨*Beams: public*⟩+≡
```
public :: beam_set_momenta
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_set_momenta (beam, p)
  type(beam_t), intent(inout) :: beam
  type(vector4_t), dimension(:), intent(in) :: p
  call interaction_set_momenta (beam%int, p)
```

```
      end subroutine beam_set_momenta
```

9.2.9 Unit tests

⟨*Beams: public*⟩+≡
```
    public :: beam_test
```

⟨*Beams: procedures*⟩+≡
```
    subroutine beam_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Beams: execute tests⟩
    end subroutine beam_test
```

Test the basic beam setup.

⟨*Beams: execute tests*⟩≡
```
    call test (beam_1, "beam_1", &
         "check basic beam setup", &
         u, results)
```

⟨*Beams: tests*⟩≡
```
    subroutine beam_1 (u)
      use os_interface
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(beam_data_t), target :: beam_data
      type(beam_t) :: beam
      real(default) :: sqrts
      type(flavor_t), dimension(2) :: flv
      type(smatrix_t), dimension(2) :: smatrix
      real(default), dimension(2) :: pol_f
      type(model_list_t) :: model_list
      type(model_t), pointer :: model

      write (u, "(A)")  "* Test output: beam_1"
      write (u, "(A)")  "*   Purpose: test basic beam setup"
      write (u, "(A)")

      write (u, "(A)")  "* Reading model file"
      write (u, "(A)")

      call reset_interaction_counter ()

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
           (var_str("SM"), var_str("SM.mdl"), os_data, model)
      call syntax_model_file_final ()

      write (u, "(A)")  "* 1: Scattering process"
      write (u, "(A)")

      sqrts = 500
```

```
   call flavor_init (flv, [1,-1], model)

   call smatrix(1)%init (2, 1)
   call smatrix(1)%set_entry (1, [1,1], (1._default, 0._default))
   pol_f(1) = 0.5_default

   !!! 2.1 version:
   ! call polarization_init_circular (pol(1), flv(1), 0.5_default)

   call smatrix(2)%init (2, 3)
   call smatrix(2)%set_entry (1, [1,1], (1._default, 0._default))
   call smatrix(2)%set_entry (2, [-1,-1], (1._default, 0._default))
   call smatrix(2)%set_entry (3, [-1,1], (1._default, 0._default))
   pol_f(2) = 1._default

   !!! 2.1 version:
   ! call polarization_init_transversal (pol(2), flv(2), 0._default, 1._default)
   call beam_data_init_sqrts (beam_data, sqrts, flv, smatrix, pol_f)
   call beam_data_write (beam_data, u)
   write (u, "(A)")
   call beam_init (beam, beam_data)
   call beam_write (beam, u)
   call beam_final (beam)
   call beam_data_final (beam_data)

   write (u, "(A)")
   write (u, "(A)")  "* 2: Decay"
   write (u, "(A)")
   call flavor_init (flv(1), 23, model)
   call smatrix(1)%init (2, 1)
   call smatrix(1)%set_entry (1, [0,0], (1._default, 0._default))
   pol_f(1) = 0.4_default

   !!! 2.1 version:
   ! call polarization_init_longitudinal (pol(1), flv(1), 0.4_default)
   call beam_data_init_decay (beam_data, flv(1:1), smatrix(1:1), pol_f(1:1))
   call beam_data_write (beam_data, u)
   write (u, "(A)")
   call beam_init (beam, beam_data)
   call beam_write (beam, u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call beam_final (beam)
   call beam_data_final (beam_data)
   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: beam_1"

end subroutine beam_1
```

Test advanced beam setup.

⟨*Beams: execute tests*⟩+≡
```
    call test (beam_2, "beam_2", &
        "beam initialization", &
        u, results)
```

⟨*Beams: tests*⟩+≡
```
    subroutine beam_2 (u)
      use os_interface
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(beam_data_t), target :: beam_data
      type(beam_t) :: beam
      real(default) :: sqrts
      type(flavor_t), dimension(2) :: flv
      integer, dimension(0) :: no_records
      type(beam_structure_t) :: beam_structure
      type(model_list_t) :: model_list
      type(model_t), pointer :: model

      write (u, "(A)")  "* Test output: beam_2"
      write (u, "(A)")  "*   Purpose: transfer beam polarization using &
          &beam structure"
      write (u, "(A)")

      write (u, "(A)")  "* Reading model file"
      write (u, "(A)")

      call reset_interaction_counter ()

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
          (var_str("SM"), var_str("SM.mdl"), os_data, model)
      call syntax_model_file_final ()

      write (u, "(A)")  "* 1: Scattering process"
      write (u, "(A)")

      sqrts = 500
      call flavor_init (flv, [1,-1], model)
      call beam_structure%init_sf (flavor_get_name (flv), no_records)

      call beam_structure%init_pol (2)

      call beam_structure%init_smatrix (1, 1)
      call beam_structure%set_sentry (1, 1, [1,1], (1._default, 0._default))

      call beam_structure%init_smatrix (2, 3)
      call beam_structure%set_sentry (2, 1, [1,1], (1._default, 0._default))
      call beam_structure%set_sentry (2, 2, [-1,-1], (1._default, 0._default))
      call beam_structure%set_sentry (2, 3, [-1,1], (1._default, 0._default))

      call beam_structure%set_pol_f ([0.5_default, 1._default])
      call beam_structure%write (u)
```

```
      write (u, *)

      call beam_data_init_structure (beam_data, beam_structure, sqrts, model)
      call beam_data_write (beam_data, u)
      write (u, *)

      call beam_init (beam, beam_data)
      call beam_write (beam, u)

      call beam_final (beam)
      call beam_data_final (beam_data)
      call beam_structure%final_pol ()
      call beam_structure%final_sf ()

      write (u, "(A)")
      write (u, "(A)")  "* 2: Decay"
      write (u, "(A)")

      call flavor_init (flv(1), 23, model)
      call beam_structure%init_sf ([flavor_get_name (flv(1))], no_records)

      call beam_structure%init_pol (1)

      call beam_structure%init_smatrix (1, 1)
      call beam_structure%set_sentry (1, 1, [0,0], (1._default, 0._default))
      call beam_structure%set_pol_f ([0.4_default])
      call beam_structure%write (u)
      write (u, *)

      call beam_data_init_structure (beam_data, beam_structure, sqrts, model)
      call beam_data_write (beam_data, u)
      write (u, "(A)")
      call beam_init (beam, beam_data)
      call beam_write (beam, u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call beam_final (beam)
      call beam_data_final (beam_data)
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: beam_2"

    end subroutine beam_2
```

Test advanced beam setup, completely arbitrary momenta.

⟨*Beams: execute tests*⟩+≡
```
    call test (beam_3, "beam_3", &
         "generic beam momenta", &
         u, results)
```

⟨*Beams: tests*⟩+≡

```
subroutine beam_3 (u)
  use os_interface
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(beam_data_t), target :: beam_data
  type(beam_t) :: beam
  type(flavor_t), dimension(2) :: flv
  integer, dimension(0) :: no_records
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(beam_structure_t) :: beam_structure
  type(vector3_t), dimension(2) :: p3
  type(vector4_t), dimension(2) :: p

  write (u, "(A)")  "* Test output: beam_3"
  write (u, "(A)")  "*   Purpose: set up beams with generic momenta"
  write (u, "(A)")

  write (u, "(A)")  "* Reading model file"
  write (u, "(A)")

  call reset_interaction_counter ()

  call syntax_model_file_init ()
  call os_data_init (os_data)
  call model_list%read_model &
       (var_str("SM"), var_str("SM.mdl"), os_data, model)
  call syntax_model_file_final ()

  write (u, "(A)")  "* 1: Scattering process"
  write (u, "(A)")

  call flavor_init (flv, [2212,2212], model)

  p3(1) = vector3_moving ([5._default, 0._default, 10._default])
  p3(2) = -vector3_moving ([1._default, 1._default, -10._default])

  call beam_structure%init_sf (flavor_get_name (flv), no_records)
  call beam_structure%set_momentum (p3 ** 1)
  call beam_structure%set_theta (polar_angle (p3))
  call beam_structure%set_phi (azimuthal_angle (p3))
  call beam_structure%write (u)
  write (u, *)

  call beam_data_init_structure (beam_data, beam_structure, 0._default, model)
  call beam_data_write (beam_data, u, verbose = .true.)
  write (u, *)

  write (u, "(1x,A)")  "Beam momenta reconstructed from LT:"
  p = beam_data%L_cm_to_lab * beam_data%p_cm
  call pacify (p, 1e-12_default)
  call vector4_write (p(1), u)
  call vector4_write (p(2), u)
```

```
    write (u, "(A)")

    call beam_init (beam, beam_data)
    call beam_write (beam, u)

    call beam_final (beam)
    call beam_data_final (beam_data)
    call beam_structure%final_sf ()
    call beam_structure%final_mom ()

    write (u, "(A)")
    write (u, "(A)")  "* 2: Decay"
    write (u, "(A)")

    call flavor_init (flv(1), 23, model)
    p3(1) = vector3_moving ([10._default, 5._default, 50._default])

    call beam_structure%init_sf ([flavor_get_name (flv(1))], no_records)
    call beam_structure%set_momentum ([p3(1) ** 1])
    call beam_structure%set_theta ([polar_angle (p3(1))])
    call beam_structure%set_phi ([azimuthal_angle (p3(1))])
    call beam_structure%write (u)
    write (u, *)

    call beam_data_init_structure (beam_data, beam_structure, 0._default, model)
    call beam_data_write (beam_data, u, verbose = .true.)
    write (u, "(A)")

    write (u, "(1x,A)")  "Beam momentum reconstructed from LT:"
    p(1) = beam_data%L_cm_to_lab * beam_data%p_cm(1)
    call pacify (p(1), 1e-12_default)
    call vector4_write (p(1), u)
    write (u, "(A)")

    call beam_init (beam, beam_data)
    call beam_write (beam, u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call beam_final (beam)
    call beam_data_final (beam_data)
    call beam_structure%final_sf ()
    call beam_structure%final_mom ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: beam_3"

  end subroutine beam_3
```

Chapter 10

Spectra and structure functions

10.1 Tools

This module contains auxiliary procedures that can be accessed by the structure function code.

⟨`sf_aux.f90`⟩≡
  ⟨*File header*⟩

  `module sf_aux`

  ⟨*Use kinds*⟩
    `use constants, only: twopi !NODEP!`
  ⟨*Use file utils*⟩
    `use unit_tests`

    `use lorentz !NODEP!`

  ⟨*Standard module head*⟩

  ⟨*SF aux: public*⟩

  ⟨*SF aux: parameters*⟩

  ⟨*SF aux: types*⟩

  `contains`

  ⟨*SF aux: procedures*⟩

  ⟨*SF aux: tests*⟩

  `end module sf_aux`

10.1.1 Momentum splitting

Let us consider first an incoming parton with momentum k and invariant mass squared $s = k^2$ that splits into two partons with momenta q, p and invariant masses $t = q^2$ and $u = p^2$. (This is an abuse of the Mandelstam notation. t is actually the momentum transfer, assuming that p is radiated and q initiates the hard process.) The energy is split among the partons such that if $E = k^0$, we have $q^0 = xE$ and $p^0 = \bar{x}E$, where $\bar{x} \equiv 1 - x$.

We define the angle θ as the polar angle of p w.r.t. the momentum axis of the incoming momentum k. Ignoring azimuthal angle, we can write the four-momenta in the basis (E, p_T, p_L) as

$$k = \begin{pmatrix} E \\ 0 \\ p \end{pmatrix}, \qquad p = \begin{pmatrix} \bar{x}E \\ \bar{x}\bar{p}\sin\theta \\ \bar{x}\bar{p}\cos\theta \end{pmatrix}, \qquad q = \begin{pmatrix} xE \\ -\bar{x}\bar{p}\sin\theta \\ p - \bar{x}\bar{p}\cos\theta \end{pmatrix}, \qquad (10.1)$$

where the first two mass-shell conditions are

$$p^2 = E^2 - s, \qquad \bar{p}^2 = E^2 - \frac{u}{\bar{x}^2}. \qquad (10.2)$$

The second condition implies that, for positive u, $\bar{x}^2 > u/E^2$, or equivalently

$$x < 1 - \sqrt{u}/E. \qquad (10.3)$$

We are interested in the third mass-shell conditions: s and u are fixed, so we need t as a function of $\cos\theta$:

$$t = -2\bar{x}\left(E^2 - p\bar{p}\cos\theta\right) + s + u. \qquad (10.4)$$

Solving for $\cos\theta$, we get

$$\cos\theta = \frac{2\bar{x}E^2 + t - s - u}{2\bar{x}p\bar{p}}. \qquad (10.5)$$

We can compute $\sin\theta$ numerically as $\sin^2\theta = 1 - \cos^2\theta$, but it is important to reexpress this in view of numerical stability. To this end, we first determine the bounds for t. The cosine must be between -1 and 1, so the bounds are

$$t_0 = -2\bar{x}\left(E^2 + p\bar{p}\right) + s + u, \qquad (10.6)$$
$$t_1 = -2\bar{x}\left(E^2 - p\bar{p}\right) + s + u. \qquad (10.7)$$

Computing $\sin^2\theta$ from $\cos\theta$ above, we observe that the numerator is a quadratic polynomial in t which has the zeros t_0 and t_1, while the common denominator is given by $(2\bar{x}p\bar{p})^2$. Hence, we can write

$$\sin^2\theta = -\frac{(t - t_0)(t - t_1)}{(2\bar{x}p\bar{p})^2} \qquad \text{and} \qquad \cos\theta = \frac{(t - t_0) + (t - t_1)}{4\bar{x}p\bar{p}}, \qquad (10.8)$$

which is free of large cancellations near $t = t_0$ or $t = t_1$.

If all is massless, i.e., $s = u = 0$, this simplifies to

$$t_0 = -4\bar{x}E^2, \qquad\qquad\qquad t_1 = 0, \qquad (10.9)$$
$$\sin^2\theta = -\frac{t}{\bar{x}E^2}\left(1 + \frac{t}{4\bar{x}E^2}\right), \qquad \cos\theta = 1 + \frac{t}{2\bar{x}E^2}. \qquad (10.10)$$

Here is the implementation. First, we define a container for the kinematical integration limits and some further data.

⟨*SF aux: public*⟩≡
```
    public :: splitting_data_t
```

⟨*SF aux: types*⟩≡
```
    type :: splitting_data_t
      private
      logical :: collinear = .false.
      real(default) :: x0 = 0
      real(default) :: x1
      real(default) :: t0
      real(default) :: t1
      real(default) :: phi0 = 0
      real(default) :: phi1 = twopi
      real(default) :: E, p, s, u, m2
      real(default) :: x, xb, pb
      real(default) :: t = 0
      real(default) :: phi = 0
     contains
     ⟨SF aux: splitting data: TBP⟩
    end type splitting_data_t
```

I/O for debugging:

⟨*SF aux: splitting data: TBP*⟩≡
```
    procedure :: write => splitting_data_write
```

⟨*SF aux: procedures*⟩≡
```
    subroutine splitting_data_write (d, unit)
      class(splitting_data_t), intent(in) :: d
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)") "Splitting data:"
      write (u, "(2x,A,L1)")  "collinear = ", d%collinear
  1   format (2x,A,1x,ES15.8)
      write (u, 1) "x0   =", d%x0
      write (u, 1) "x    =", d%x
      write (u, 1) "xb   =", d%xb
      write (u, 1) "x1   =", d%x1
      write (u, 1) "t0   =", d%t0
      write (u, 1) "t    =", d%t
      write (u, 1) "t1   =", d%t1
      write (u, 1) "phi0 =", d%phi0
      write (u, 1) "phi  =", d%phi
      write (u, 1) "phi1 =", d%phi1
      write (u, 1) "E    =", d%E
      write (u, 1) "p    =", d%p
      write (u, 1) "pb   =", d%pb
      write (u, 1) "s    =", d%s
      write (u, 1) "u    =", d%u
      write (u, 1) "m2   =", d%m2
    end subroutine splitting_data_write
```

10.1.2 Constant data

This is the initializer for the data. The input consists of the incoming momentum, its invariant mass squared, and the invariant mass squared of the radiated particle. $m2$ is the *physical* mass squared of the outgoing particle. The t bounds depend on the chosen x value and cannot be determined yet.

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: init => splitting_data_init
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_data_init (d, k, mk2, mr2, mo2, collinear)
      class(splitting_data_t), intent(out) :: d
      type(vector4_t), intent(in) :: k
      real(default), intent(in) :: mk2, mr2, mo2
      logical, intent(in), optional :: collinear
      if (present (collinear))  d%collinear = collinear
      d%E = energy (k)
      d%x1 = 1 - sqrt (max (mr2, 0._default)) / d%E
      d%p = sqrt (d%E**2 - mk2)
      d%s = mk2
      d%u = mr2
      d%m2 = mo2
    end subroutine splitting_data_init
```

Retrieve the x bounds, if needed for x sampling. Generating an x value is done by the caller, since this is the part that depends on the nature of the structure function.

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: get_x_bounds => splitting_get_x_bounds
```

⟨*SF aux: procedures*⟩+≡
```
    function splitting_get_x_bounds (d) result (x)
      class(splitting_data_t), intent(in) :: d
      real(default), dimension(2) :: x
      x = [ d%x0, d%x1 ]
    end function splitting_get_x_bounds
```

Now set the momentum fraction and compute t_0 and t_1.

[The calculation of t_1 is subject to numerical problems. The exact formula is ($s = m_i^2$, $u = m_r^2$)

$$t_1 = -2\bar{x}E^2 + m_i^2 + m_r^2 + 2\bar{x}\sqrt{E^2 - m_i^2}\,\sqrt{E^2 - m_r^2/\bar{x}^2}. \qquad (10.11)$$

The structure-function paradigm is useful only if $E \gg m_i, m_r$. In a Taylor expansion for large E, the leading term cancels. The expansion of the square roots (to subleading order) yields

$$t_1 = xm_i^2 - \frac{x}{\bar{x}}m_r^2. \qquad (10.12)$$

There are two cases of interest: $m_i = m_o$ and $m_r = 0$,

$$t_1 = xm_o^2 \qquad (10.13)$$

986

and $m_i = m_r$ and $m_o = 0$,

$$t_1 = -\frac{x^2}{\bar{x}}m_i^2. \tag{10.14}$$

In both cases, $t_1 \leq m_o^2$.]

That said, it turns out that taking the t_1 evaluation at face value leads to less problems than the approximation. We express the angles in terms of $t - t_0$ and $t - t_1$. Numerical noise in t_1 can then be tolerated.

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: set_t_bounds => splitting_set_t_bounds
```

⟨*SF aux: procedures*⟩+≡
```
    elemental subroutine splitting_set_t_bounds (d, x, xb)
      class(splitting_data_t), intent(inout) :: d
      real(default), intent(in), optional :: x, xb
      real(default) :: tp, tm
      if (present (x))  d%x = x
      if (present (xb)) d%xb = xb
      if (d%xb /= 0) then
         d%pb = sqrt (max (d%E**2 - d%u / d%xb**2, 0._default))
      else
         d%pb = 0
      end if
      tp = -2 * d%xb * d%E**2 + d%s + d%u
      tm = -2 * d%xb * d%p * d%pb
      d%t0 = tp + tm
      d%t1 = tp - tm
      d%t = d%t1
    end subroutine splitting_set_t_bounds
```

10.1.3   Sampling recoil

Compute a value for the momentum transfer t, using a random number r. We assume a logarithmic distribution for $t - m^2$, corresponding to the propagator $1/(t - m^2)$ with the physical mass m for the outgoing particle. Optionally, we can narrow the kinematical bounds.

If all three masses in the splitting vanish, the upper limit for t is zero. In that case, the t value is set to zero and the splitting will be collinear.

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: sample_t => splitting_sample_t
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_sample_t (d, r, t0, t1)
      class(splitting_data_t), intent(inout) :: d
      real(default), intent(in) :: r
      real(default), intent(in), optional :: t0, t1
      real(default) :: tt0, tt1, tt0m, tt1m
      if (d%collinear) then
         d%t = d%t1
      else
         tt0 = d%t0;  if (present (t0))  tt0 = max (t0, tt0)
         tt1 = d%t1;  if (present (t1))  tt1 = min (t1, tt1)
         tt0m = tt0 - d%m2
```

```
            tt1m = tt1 - d%m2
            if (tt0m < 0 .and. tt1m < 0 .and. abs(tt0m) > &
                  epsilon(tt0m) .and. abs(tt1m) > epsilon(tt0m)) then
               d%t = d%m2 + tt0m * exp (r * log (tt1m / tt0m))
            else
               d%t = tt1
            end if
         end if
      end subroutine splitting_sample_t
```

The inverse operation: Given t, we recover the value of r that would have produced this value.

⟨SF aux: splitting data: TBP⟩+≡
```
      procedure :: inverse_t => splitting_inverse_t
```

⟨SF aux: procedures⟩+≡
```
      subroutine splitting_inverse_t (d, r, t0, t1)
        class(splitting_data_t), intent(in) :: d
        real(default), intent(out) :: r
        real(default), intent(in), optional :: t0, t1
        real(default) :: tt0, tt1, tt0m, tt1m
        if (d%collinear) then
           r = 0
        else
           tt0 = d%t0;  if (present (t0))  tt0 = max (t0, tt0)
           tt1 = d%t1;  if (present (t1))  tt1 = min (t1, tt1)
           tt0m = tt0 - d%m2
           tt1m = tt1 - d%m2
           if (tt0m < 0 .and. tt1m < 0) then
               r = log ((d%t - d%m2) / tt0m) / log (tt1m / tt0m)
           else
               r = 0
           end if
        end if
      end subroutine splitting_inverse_t
```

This is trivial, but provided for convenience:

⟨SF aux: splitting data: TBP⟩+≡
```
      procedure :: sample_phi => splitting_sample_phi
```

⟨SF aux: procedures⟩+≡
```
      subroutine splitting_sample_phi (d, r)
        class(splitting_data_t), intent(inout) :: d
        real(default), intent(in) :: r
        if (d%collinear) then
           d%phi = 0
        else
           d%phi = (1-r) * d%phi0 + r * d%phi1
        end if
      end subroutine splitting_sample_phi
```

Inverse:

⟨SF aux: splitting data: TBP⟩+≡
```
      procedure :: inverse_phi => splitting_inverse_phi
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_inverse_phi (d, r)
      class(splitting_data_t), intent(in) :: d
      real(default), intent(out) :: r
      if (d%collinear) then
         r = 0
      else
         r = (d%phi - d%phi0) / (d%phi1 - d%phi0)
      end if
    end subroutine splitting_inverse_phi
```

10.1.4 Splitting

In this function, we actually perform the splitting. The incoming momentum k is split into (if no recoil) $q_1 = (1 - x)k$ and $q_2 = xk$.

Apart from the splitting data, we need the incoming momentum k, the momentum transfer t, and the azimuthal angle ϕ. The momentum fraction x is already known here.

Alternatively, we can split without recoil. The azimuthal angle is irrelevant, and the momentum transfer is always equal to the upper limit t_1, so the polar angle is zero. Obviously, if there are nonzero masses it is not possible to keep both energy-momentum conservation and at the same time all particles on shell. We choose for dropping the on-shell condition here.

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: split_momentum => splitting_split_momentum
```

⟨*SF aux: procedures*⟩+≡
```
    function splitting_split_momentum (d, k) result (q)
      class(splitting_data_t), intent(in) :: d
      type(vector4_t), dimension(2) :: q
      type(vector4_t), intent(in) :: k
      real(default) :: st2, ct2, st, ct, cp, sp
      type(lorentz_transformation_t) :: rot
      real(default) :: tt0, tt1, den
      type(vector3_t) :: kk, q1, q2
      if (d%collinear) then
         if (d%s == 0 .and. d%u == 0) then
            q(1) = d%xb * k
            q(2) = d%x * k
         else
            kk = space_part (k)
            q1 = d%xb * (d%pb / d%p) * kk
            q2 = kk - q1
            q(1) = vector4_moving (d%xb * d%E, q1)
            q(2) = vector4_moving (d%x * d%E, q2)
         end if
      else
         den = 2 * d%xb * d%p * d%pb
         tt0 = max (d%t - d%t0, 0._default)
         tt1 = min (d%t - d%t1, 0._default)
         if (den**2 <= epsilon(den)) then
            st2 = 1
```

```
        else
            st2 = - (tt0 * tt1) / den ** 2
        end if
        if (st2 > 1) then
            st2 = 1
        end if
        ct2 = 1 - st2
        st = sqrt (max (st2, 0._default))
        ct = sqrt (max (ct2, 0._default))
        sp = sin (d%phi)
        cp = cos (d%phi)
        rot = rotation_to_2nd (3, space_part (k))
        q1 = vector3_moving (d%xb * d%pb * [st * cp, st * sp, ct])
        q2 = vector3_moving (d%p, 3) - q1
        q(1) = rot * vector4_moving (d%xb * d%E, q1)
        q(2) = rot * vector4_moving (d%x * d%E, q2)
      end if
    end function splitting_split_momentum
```

Momenta generated by splitting will in general be off-shell. They are on-shell only if they are collinear and massless. This subroutine puts them on shell by brute force, violating either momentum or energy conservation. The direction of three-momentum is always retained.

If the energy is below mass shell, we return a zero momentum.

⟨*SF aux: parameters*⟩≡
```
    integer, parameter, public :: KEEP_ENERGY = 0, KEEP_MOMENTUM = 1
```

⟨*SF aux: public*⟩+≡
```
    public :: on_shell
```

⟨*SF aux: procedures*⟩+≡
```
    elemental subroutine on_shell (p, m2, keep)
      type(vector4_t), intent(inout) :: p
      real(default), intent(in) :: m2
      integer, intent(in) :: keep
      real(default) :: E, E2, pn
      select case (keep)
      case (KEEP_ENERGY)
         E = energy (p)
         E2 = E ** 2
         if (E2 >= m2) then
            pn = sqrt (E2 - m2)
            p = vector4_moving (E, pn * direction (space_part (p)))
         else
            p = vector4_null
         end if
      case (KEEP_MOMENTUM)
         E = sqrt (space_part (p) ** 2 + m2)
         p = vector4_moving (E, space_part (p))
      end select
    end subroutine on_shell
```

10.1.5 Recovering the splitting

This is the inverse problem. We have on-shell momenta and want to deduce the splitting parameters x, t, and ϕ.

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: recover => splitting_recover
```

⟨*SF aux: procedures*⟩+≡
```
    subroutine splitting_recover (d, k, q, keep)
      class(splitting_data_t), intent(inout) :: d
      type(vector4_t), intent(in) :: k, q
      integer, intent(in) :: keep
      type(lorentz_transformation_t) :: rot
      type(vector4_t) :: q0, k0
      real(default) :: p1, p2, p3, pt2, pp2, pl
      real(default) :: aux, den, norm
      real(default) :: st2, ct2, ct
      rot = inverse (rotation_to_2nd (3, space_part (k)))
      q0 = rot * q
      p1 = vector4_get_component (q0, 1)
      p2 = vector4_get_component (q0, 2)
      p3 = vector4_get_component (q0, 3)
      pt2 = p1 ** 2 + p2 ** 2
      pp2 = p1 ** 2 + p2 ** 2 + p3 ** 2
      pl = abs (p3)
      k0 = vector4_moving (d%E, d%p, 3)
      select case (keep)
      case (KEEP_ENERGY)
         d%x = energy (q0) / d%E
         d%xb = 1 - d%x
         call d%set_t_bounds ()
         if (.not. d%collinear) then
            aux = (d%xb * d%pb) ** 2 * pp2 - d%p ** 2 * pt2
            den = d%p ** 2 - (d%xb * d%pb) ** 2
            if (aux >= 0 .and. den > 0) then
               norm = (d%p * pl + sqrt (aux)) / den
            else
               norm = 1
            end if
         end if
      case (KEEP_MOMENTUM)
         d%xb = sqrt (space_part (k0 - q0) ** 2 + d%u) / d%E
         d%x = 1 - d%xb
         call d%set_t_bounds ()
         norm = 1
      end select
      if (d%collinear) then
         d%t = d%t1
         d%phi = 0
      else
         if ((d%xb * d%pb * norm)**2 < epsilon(d%xb)) then
            st2 = 1
         else
            st2 = pt2 / (d%xb * d%pb * norm ) ** 2
         end if
```

```
        if (st2 > 1) then
            st2 = 1
        end if
        ct2 = 1 - st2
        ct = sqrt (max (ct2, 0._default))
        if (ct /= -1) then
            d%t = d%t1 - 2 * d%xb * d%p * d%pb * st2 / (1 + ct)
        else
            d%t = d%t0
        end if
        if (p1 /= 0 .or. p2 /= 0) then
            d%phi = atan2 (-p2, -p1)
        else
            d%phi = 0
        end if
    end if
end subroutine splitting_recover
```

10.1.6   Extract data

⟨*SF aux: splitting data: TBP*⟩+≡
```
    procedure :: get_x => splitting_get_x
```

⟨*SF aux: procedures*⟩+≡
```
    function splitting_get_x (sd) result (x)
      class(splitting_data_t), intent(in) :: sd
      real(default) :: x
      x = sd%x
    end function splitting_get_x
```

10.1.7   Unit tests

⟨*SF aux: public*⟩+≡
```
    public :: sf_aux_test
```

⟨*SF aux: tests*⟩≡
```
    subroutine sf_aux_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF aux: execute tests⟩
    end subroutine sf_aux_test
```

Momentum splitting: massless radiation

Compute momentum splitting for generic kinematics. It turns out that for $x = 0.5$, where $t - m^2$ is the geometric mean between its upper and lower bounds (this can be directly seen from the logarithmic distribution in the function `sample_t` for $r \equiv x = 1 - x = 0.5$), we arrive at an exact number $t = -0.15$ for the given input values.

⟨*SF aux: execute tests*⟩≡

```
        call test (sf_aux_1, "sf_aux_1", &
             "massless radiation", &
             u, results)
```
⟨*SF aux: tests*⟩+≡
```
      subroutine sf_aux_1 (u)
        integer, intent(in) :: u
        type(splitting_data_t) :: sd
        type(vector4_t) :: k
        type(vector4_t), dimension(2) :: q, q0
        real(default) :: E, mk, mp, mq
        real(default) :: x, r1, r2, r1o, r2o
        real(default) :: k2, q0_2, q1_2, q2_2

        write (u, "(A)")  "* Test output: sf_aux_1"
        write (u, "(A)")  "*   Purpose: compute momentum splitting"
        write (u, "(A)")  "             (massless radiated particle)"
        write (u, "(A)")

        E = 1
        mk = 0.3_default
        mp = 0
        mq = mk

        k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
        k2 = k ** 2;  call pacify (k2, 1e-10_default)

        x = 0.6_default
        r1 = 0.5_default
        r2 = 0.125_default

        write (u, "(A)")  "* (1) Non-collinear setup"
        write (u, "(A)")

        call sd%init (k, mk**2, mp**2, mq**2)
        call sd%set_t_bounds (x, 1 - x)
        call sd%sample_t (r1)
        call sd%sample_phi (r2)

        call sd%write (u)

        q = sd%split_momentum (k)
        q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
        q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

        write (u, "(A)")
        write (u, "(A)")  "Incoming momentum k ="
        call vector4_write (k, u)
        write (u, "(A)")
        write (u, "(A)")  "Outgoing momentum sum p + q ="
        call vector4_write (sum (q), u)
        write (u, "(A)")
        write (u, "(A)")  "Radiated momentum p ="
        call vector4_write (q(1), u)
        write (u, "(A)")
```

993

```
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q0_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t
```

```
call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q0_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
```

```
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"
```

```
q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q0_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
```

```
      call vector4_write (q0(2), u)
      write (u, "(A)")

      write (u, "(A)")  "Compare: mo^2"
      q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
      write (u, "(2(1x,F11.8))")  sd%m2, q0_2
      write (u, "(A)")

      write (u, "(A)")  "* Recover parameters from outgoing momentum"
      write (u, "(A)")

      call sd%init (k, mk**2, mp**2, mq**2)
      call sd%recover (k, q0(2), KEEP_MOMENTUM)

      write (u, "(A)")  "Compare: x"
      write (u, "(2(1x,F11.8))")  x, sd%x
      write (u, "(A)")  "Compare: t"
      write (u, "(2(1x,F11.8))")  q2_2, sd%t

      write (u, "(A)")
      call sd%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_aux_1"

    end subroutine sf_aux_1
```

Momentum splitting: massless parton

Compute momentum splitting for generic kinematics. It turns out that for $x = 0.5$, where $t - m^2$ is the geometric mean between its upper and lower bounds, we arrive at an exact number $t = -0.36$ for the given input values.

⟨*SF aux: execute tests*⟩+≡
```
    call test (sf_aux_2, "sf_aux_2", &
         "massless parton", &
         u, results)
```

⟨*SF aux: tests*⟩+≡
```
    subroutine sf_aux_2 (u)
      integer, intent(in) :: u
      type(splitting_data_t) :: sd
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q, q0
      real(default) :: E, mk, mp, mq
      real(default) :: x, r1, r2, r1o, r2o
      real(default) :: k2, q02_2, q1_2, q2_2

      write (u, "(A)")  "* Test output: sf_aux_2"
      write (u, "(A)")  "*   Purpose: compute momentum splitting"
      write (u, "(A)")  "             (massless outgoing particle)"
      write (u, "(A)")

      E = 1
```

```
mk = 0.3_default
mp = mk
mq = 0

k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
k2 = k ** 2;  call pacify (k2, 1e-10_default)

x = 0.6_default
r1 = 0.5_default
r2 = 0.125_default

write (u, "(A)")  "* (1) Non-collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%sample_t (r1)
call sd%sample_phi (r2)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)
```

```
write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t


call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"
```

```
q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)
```

```
                    call sd%write (u)

                    q = sd%split_momentum (k)
                    q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
                    q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

                    write (u, "(A)")
                    write (u, "(A)")  "Incoming momentum k ="
                    call vector4_write (k, u)
                    write (u, "(A)")
                    write (u, "(A)")  "Outgoing momentum sum p + q ="
                    call vector4_write (sum (q), u)
                    write (u, "(A)")
                    write (u, "(A)")  "Radiated momentum p ="
                    call vector4_write (q(1), u)
                    write (u, "(A)")
                    write (u, "(A)")  "Outgoing momentum q ="
                    call vector4_write (q(2), u)
                    write (u, "(A)")

                    write (u, "(A)")  "Compare: s"
                    write (u, "(2(1x,F11.8))")  sd%s, k2

                    write (u, "(A)")  "Compare: t"
                    write (u, "(2(1x,F11.8))")  sd%t, q2_2

                    write (u, "(A)")  "Compare: u"
                    write (u, "(2(1x,F11.8))")  sd%u, q1_2

                    write (u, "(A)")  "Compare: x"
                    write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

                    write (u, "(A)")  "Compare: 1-x"
                    write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

                    write (u, "(A)")
                    write (u, "(A)")  "* Project on-shell (keep energy)"

                    q0 = q
                    call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

                    write (u, "(A)")
                    write (u, "(A)")  "Incoming momentum k ="
                    call vector4_write (k, u)
                    write (u, "(A)")
                    write (u, "(A)")  "Outgoing momentum sum p + q ="
                    call vector4_write (sum (q0), u)
                    write (u, "(A)")
                    write (u, "(A)")  "Radiated momentum p ="
                    call vector4_write (q0(1), u)
                    write (u, "(A)")
                    write (u, "(A)")  "Outgoing momentum q ="
                    call vector4_write (q0(2), u)
                    write (u, "(A)")
```

```
write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
```

```
    write (u, "(2(1x,F11.8))")  x, sd%x
    write (u, "(A)")  "Compare: t"
    write (u, "(2(1x,F11.8))")  q2_2, sd%t

    write (u, "(A)")
    call sd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_aux_2"

  end subroutine sf_aux_2
```

Momentum splitting: all massless

Compute momentum splitting for massless kinematics. In the non-collinear
case, we need a lower cutoff for $|t|$, otherwise a logarithmic distribution is not
possible.

⟨*SF aux: execute tests*⟩+≡
```
    call test (sf_aux_3, "sf_aux_3", &
         "massless parton", &
         u, results)
```

⟨*SF aux: tests*⟩+≡
```
    subroutine sf_aux_3 (u)
      integer, intent(in) :: u
      type(splitting_data_t) :: sd
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q, q0
      real(default) :: E, mk, mp, mq, qmin, qmax
      real(default) :: x, r1, r2, r1o, r2o
      real(default) :: k2, q02_2, q1_2, q2_2

      write (u, "(A)")  "* Test output: sf_aux_3"
      write (u, "(A)")  "*   Purpose: compute momentum splitting"
      write (u, "(A)")  "               (all massless, q cuts)"
      write (u, "(A)")

      E = 1
      mk = 0
      mp = 0
      mq = 0
      qmin = 1e-2_default
      qmax = 1e0_default

      k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
      k2 = k ** 2;  call pacify (k2, 1e-10_default)

      x = 0.6_default
      r1 = 0.5_default
      r2 = 0.125_default

      write (u, "(A)")  "* (1) Non-collinear setup"
      write (u, "(A)")
```

```
call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%sample_t (r1, t1 = - qmin ** 2, t0 = - qmax **2)
call sd%sample_phi (r2)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
```

```
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t


call sd%inverse_t (r1o, t1 = - qmin ** 2, t0 = - qmax **2)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
```

```
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

call sd%inverse_t (r1o, t1 = - qmin ** 2, t0 = - qmax **2)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
```

```
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)
```

```
    write (u, "(A)")  "Compare: x"
    write (u, "(2(1x,F11.8))")  x, sd%x
    write (u, "(A)")  "Compare: t"
    write (u, "(2(1x,F11.8))")  q2_2, sd%t

    write (u, "(A)")
    call sd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Project on-shell (keep momentum)"

    q0 = q
    call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

    write (u, "(A)")
    write (u, "(A)")  "Incoming momentum k ="
    call vector4_write (k, u)
    write (u, "(A)")
    write (u, "(A)")  "Outgoing momentum sum p + q ="
    call vector4_write (sum (q0), u)
    write (u, "(A)")
    write (u, "(A)")  "Radiated momentum p ="
    call vector4_write (q0(1), u)
    write (u, "(A)")
    write (u, "(A)")  "Outgoing momentum q ="
    call vector4_write (q0(2), u)
    write (u, "(A)")

    write (u, "(A)")  "Compare: mo^2"
    q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
    write (u, "(2(1x,F11.8))")  sd%m2, q02_2
    write (u, "(A)")

    write (u, "(A)")  "* Recover parameters from outgoing momentum"
    write (u, "(A)")

    call sd%init (k, mk**2, mp**2, mq**2)
    call sd%set_t_bounds (x, 1 - x)
    call sd%recover (k, q0(2), KEEP_MOMENTUM)

    write (u, "(A)")  "Compare: x"
    write (u, "(2(1x,F11.8))")  x, sd%x
    write (u, "(A)")  "Compare: t"
    write (u, "(2(1x,F11.8))")  q2_2, sd%t

    write (u, "(A)")
    call sd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_aux_3"

  end subroutine sf_aux_3
```

10.2 Mappings for structure functions

In this module, we provide a wrapper for useful mappings of the unit (hyper-)square that we can apply to a set of structure functions.

In some cases it is useful, or even mandatory, to map the MC input parameters nontrivially onto a set of structure functions for the two beams. In all cases considered here, instead of $x_1, x_2, \ldots$ as parameters for the beams, we generate one parameter that is equal, or related to, the product $x_1 x_2 \cdots$ (so it directly corresponds to $\sqrt{s}$). The other parameters describe the distribution of energy (loss) between beams and radiations.

⟨`sf_mappings.f90`⟩≡
  ⟨*File header*⟩

  ```
  module sf_mappings
  ```

  ⟨*Use kinds*⟩
  ```
    use kinds, only: double !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use limits, only: FMT_12, FMT_13, FMT_14, FMT_15, FMT_16 !NODEP!
    use diagnostics !NODEP!
    use constants, only: pi !NODEP!
    use unit_tests
  ```

  ⟨*Standard module head*⟩

  ⟨*SF mappings: public*⟩

  ⟨*SF mappings: parameters*⟩

  ⟨*SF mappings: types*⟩

  ⟨*SF mappings: interfaces*⟩

  ```
  contains
  ```

  ⟨*SF mappings: procedures*⟩

  ⟨*SF mappings: tests*⟩

  ```
  end module sf_mappings
  ```

10.2.1 Base type

First, we define an abstract base type for the mapping. In all cases we need to store the indices of the parameters on which the mapping applies. Additional parameters can be stored in the extensions of this type.

⟨*SF mappings: public*⟩≡
  ```
      public :: sf_mapping_t
  ```

⟨*SF mappings: types*⟩≡
  ```
      type, abstract :: sf_mapping_t
         integer, dimension(:), allocatable :: i
       contains
  ```

⟨*SF mappings: sf mapping: TBP*⟩
```
end type sf_mapping_t
```

The output routine is deferred:

⟨*SF mappings: sf mapping: TBP*⟩≡
```
procedure (sf_mapping_write), deferred :: write
```

⟨*SF mappings: interfaces*⟩≡
```
abstract interface
   subroutine sf_mapping_write (object, unit)
     import
     class(sf_mapping_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine sf_mapping_write
end interface
```

Initializer for the base type. The array of parameter indices is allocated but initialized to zero.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
procedure :: base_init => sf_mapping_base_init
```

⟨*SF mappings: procedures*⟩≡
```
subroutine sf_mapping_base_init (mapping, n_par)
  class(sf_mapping_t), intent(out) :: mapping
  integer, intent(in) :: n_par
  allocate (mapping%i (n_par))
  mapping%i = 0
end subroutine sf_mapping_base_init
```

Set an index value.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
procedure :: set_index => sf_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_mapping_set_index (mapping, j, i)
  class(sf_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
end subroutine sf_mapping_set_index
```

Return the dimensionality, i.e., the number of parameters.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
procedure :: get_n_dim => sf_mapping_get_n_dim
```

⟨*SF mappings: procedures*⟩+≡
```
function sf_mapping_get_n_dim (mapping) result (n)
  class(sf_mapping_t), intent(in) :: mapping
  integer :: n
  n = size (mapping%i)
end function sf_mapping_get_n_dim
```

Computation: the values `p` are the input parameters, the values `r` are the output parameters. The values `rb` are defined as $\bar{r} = 1 - r$, but provided explicitly. They allow us to avoid numerical problems near $r = 1$.

The extra parameter `x_free` indicates that the total energy has already been renormalized by this factor. We have to take such a factor into account in a resonance or on-shell mapping.

The Jacobian is `f`. We modify only the two parameters indicated by the indices `i`.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
    procedure (sf_mapping_compute), deferred :: compute
```

⟨*SF mappings: interfaces*⟩+≡
```
    abstract interface
      subroutine sf_mapping_compute (mapping, r, rb, f, p, pb, x_free)
        import
        class(sf_mapping_t), intent(inout) :: mapping
        real(default), dimension(:), intent(out) :: r, rb
        real(default), intent(out) :: f
        real(default), dimension(:), intent(in) :: p, pb
        real(default), intent(inout), optional :: x_free
      end subroutine sf_mapping_compute
    end interface
```

The inverse mapping. Use `r` and/or `rb` to reconstruct `p` and also compute `f`.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
    procedure (sf_mapping_inverse), deferred :: inverse
```

⟨*SF mappings: interfaces*⟩+≡
```
    abstract interface
      subroutine sf_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
        import
        class(sf_mapping_t), intent(inout) :: mapping
        real(default), dimension(:), intent(in) :: r, rb
        real(default), intent(out) :: f
        real(default), dimension(:), intent(out) :: p, pb
        real(default), intent(inout), optional :: x_free
      end subroutine sf_mapping_inverse
    end interface
```

10.2.2 Methods for self-tests

This is a shorthand for: inject parameters, compute the mapping, display results, compute the inverse, display again. We provide an output format for the parameters and, optionally, a different output format for the Jacobians.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
    procedure :: check => sf_mapping_check
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_mapping_check (mapping, u, p_in, pb_in, fmt_p, fmt_f)
      class(sf_mapping_t), intent(inout) :: mapping
      integer, intent(in) :: u
      real(default), dimension(:), intent(in) :: p_in, pb_in
```

```
      character(*), intent(in) :: fmt_p
      character(*), intent(in), optional :: fmt_f
      real(default), dimension(size(p_in)) :: p, pb, r, rb
      real(default) :: f, tolerance
      tolerance = 1.5E-17
      p = p_in
      pb= pb_in
      call mapping%compute (r, rb, f, p, pb)
      call pacify (p, tolerance)
      call pacify (pb, tolerance)
      call pacify (r, tolerance)
      call pacify (rb, tolerance)
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "p =", p
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "pb=", pb
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "r =", r
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "rb=", rb
      if (present (fmt_f)) then
         write (u, "(3x,A,9(1x," // fmt_f // "))")  "f =", f
      else
         write (u, "(3x,A,9(1x," // fmt_p // "))")  "f =", f
      end if
      write (u, *)
      call mapping%inverse (r, rb, f, p, pb)
      call pacify (p, tolerance)
      call pacify (pb, tolerance)
      call pacify (r, tolerance)
      call pacify (rb, tolerance)
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "p =", p
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "pb=", pb
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "r =", r
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "rb=", rb
      if (present (fmt_f)) then
         write (u, "(3x,A,9(1x," // fmt_f // "))")  "f =", f
      else
         write (u, "(3x,A,9(1x," // fmt_p // "))")  "f =", f
      end if
      write (u, *)
      write (u, "(3x,A,9(1x," // fmt_p // "))")  "*r=", product (r)
    end subroutine sf_mapping_check
```

This is a consistency check for the self-tests: the integral over the unit square
should be unity. We estimate this by a simple binning and adding up the values;
this should be sufficient for a self-test.

The argument is the requested number of sampling points. We take the
square root for binning in both dimensions, so the precise number might be
different.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
    procedure :: integral => sf_mapping_integral
```

⟨*SF mappings: procedures*⟩+≡
```
    function sf_mapping_integral (mapping, n_calls) result (integral)
      class(sf_mapping_t), intent(inout) :: mapping
      integer, intent(in) :: n_calls
      real(default) :: integral
```

```
integer :: n_dim, n_bin, k
real(default), dimension(:), allocatable :: p, pb, r, rb
integer, dimension(:), allocatable :: ii
real(default) :: dx, f, s

n_dim = mapping%get_n_dim ()
allocate (p (n_dim))
allocate (pb(n_dim))
allocate (r (n_dim))
allocate (rb(n_dim))
allocate (ii(n_dim))
n_bin = nint (real (n_calls, default) ** (1._default / n_dim))
dx = 1._default / n_bin
s = 0
ii = 1

SAMPLE: do
   do k = 1, n_dim
      p(k)  = ii(k) * dx - dx/2
      pb(k) = (n_bin - ii(k)) * dx + dx/2
   end do
   call mapping%compute (r, rb, f, p, pb)
   s = s + f
   INCR: do k = 1, n_dim
      ii(k) = ii(k) + 1
      if (ii(k) <= n_bin) then
         exit INCR
      else if (k < n_dim) then
         ii(k) = 1
      else
         exit SAMPLE
      end if
   end do INCR
end do SAMPLE

integral = s / real (n_bin, default) ** n_dim

end function sf_mapping_integral
```

10.2.3 Implementation: standard mapping

This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio.

⟨*SF mappings: public*⟩+≡
```
public :: sf_s_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
type, extends (sf_mapping_t) :: sf_s_mapping_t
  logical :: power_set = .false.
  real(default) :: power = 1
 contains
```
⟨*SF mappings: sf standard mapping: TBP*⟩
```
end type sf_s_mapping_t
```

1014

Output.

⟨*SF mappings: sf standard mapping: TBP*⟩≡
```
procedure :: write => sf_s_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_s_mapping_write (object, unit)
  class(sf_s_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "('(',I0,',',I0,')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A)")  ": standard (", object%power, ")"
end subroutine sf_s_mapping_write
```

Initialize: index pair and power parameter.

⟨*SF mappings: sf standard mapping: TBP*⟩+≡
```
procedure :: init => sf_s_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_s_mapping_init (mapping, power)
  class(sf_s_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: power
  call mapping%base_init (2)
  if (present (power)) then
     mapping%power_set = .true.
     mapping%power = power
  end if
end subroutine sf_s_mapping_init
```

Apply mapping.

⟨*SF mappings: sf standard mapping: TBP*⟩+≡
```
procedure :: compute => sf_s_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_s_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_s_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2
  integer :: j
  if (mapping%power_set) then
     call map_unit_square (r2, f, p(mapping%i), mapping%power)
  else
     call map_unit_square (r2, f, p(mapping%i))
  end if
  r = p
  rb= pb
  do j = 1, 2
```

```
        r (mapping%i(j)) = r2(j)
        rb(mapping%i(j)) = 1 - r2(j)
      end do
    end subroutine sf_s_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf standard mapping: TBP*⟩+≡
```
    procedure :: inverse => sf_s_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_s_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_s_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: p2
      integer :: j
      if (mapping%power_set) then
         call map_unit_square_inverse (r(mapping%i), f, p2, mapping%power)
      else
         call map_unit_square_inverse (r(mapping%i), f, p2)
      end if
      p = r
      pb= rb
      do j = 1, 2
         p (mapping%i(j)) = p2(j)
         pb(mapping%i(j)) = 1 - p2(j)
      end do
    end subroutine sf_s_mapping_inverse
```

10.2.4 Implementation: resonance pair mapping

This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio, then it maps p_1 to itself according to a Breit-Wigner shape, i.e., a flat prior distribution in p_1 results in a Breit-Wigner distribution. Mass and width of the BW are rescaled by the energy, thus dimensionless fractions.

⟨*SF mappings: public*⟩+≡
```
    public :: sf_res_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
    type, extends (sf_mapping_t) :: sf_res_mapping_t
      real(default) :: m = 0
      real(default) :: w = 0
     contains
     ⟨SF mappings: sf resonance mapping: TBP⟩
    end type sf_res_mapping_t
```

Output.

⟨*SF mappings: sf resonance mapping: TBP*⟩≡
```
    procedure :: write => sf_res_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_write (object, unit)
  class(sf_res_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
      write (u, "('(',I0,',',I0,')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,', ',F7.5,A)")  ": resonance (", object%m, object%w, ")"
end subroutine sf_res_mapping_write
```

Initialize: index pair and dimensionless mass and width parameters.

⟨*SF mappings: sf resonance mapping: TBP*⟩+≡
```
procedure :: init => sf_res_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_init (mapping, m, w)
  class(sf_res_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: m, w
  call mapping%base_init (2)
  mapping%m = m
  mapping%w = w
end subroutine sf_res_mapping_init
```

Apply mapping.

⟨*SF mappings: sf resonance mapping: TBP*⟩+≡
```
procedure :: compute => sf_res_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_res_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, p2
  real(default) :: fbw, f2, p1m
  integer :: j
  p2 = p(mapping%i)
  call map_breit_wigner &
      (p1m, fbw, p2(1), mapping%m, mapping%w, x_free)
  call map_unit_square (r2, f2, [p1m, p2(2)])
  f = fbw * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_res_mapping_compute
```

1017

Apply inverse.

⟨SF mappings: sf resonance mapping: TBP⟩+≡
```
procedure :: inverse => sf_res_mapping_inverse
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_res_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_res_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: p2
  real(default) :: fbw, f2, p1m
  call map_unit_square_inverse (r(mapping%i), f2, p2)
  call map_breit_wigner_inverse &
       (p2(1), fbw, p1m, mapping%m, mapping%w, x_free)
  p = r
  pb= rb
  p (mapping%i(1)) = p1m
  pb(mapping%i(1)) = 1 - p1m
  p (mapping%i(2)) = p2(2)
  pb(mapping%i(2)) = 1 - p2(2)
  f = fbw * f2
end subroutine sf_res_mapping_inverse
```

10.2.5   Implementation: on-shell mapping

This is a degenerate version of the unit-square mapping where the product $r_1 r_2$ is constant. This product is given by the rescaled squared mass. We introduce an artificial first parameter p_1 to keep the counting, but nothing depends on it. The second parameter is the same p_2 as for the standard unit-square mapping for $\alpha = 1$, it parameterizes the ratio of r_1 and r_2.

⟨SF mappings: public⟩+≡
```
public :: sf_os_mapping_t
```

⟨SF mappings: types⟩+≡
```
type, extends (sf_mapping_t) :: sf_os_mapping_t
  real(default) :: m = 0
  real(default) :: lm2 = 0
 contains
```
⟨SF mappings: sf on-shell mapping: TBP⟩
```
end type sf_os_mapping_t
```

Output.

⟨SF mappings: sf on-shell mapping: TBP⟩≡
```
procedure :: write => sf_os_mapping_write
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_os_mapping_write (object, unit)
  class(sf_os_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
```

```
    write (u, "(1x,A)", advance="no")  "map"
    if (any (object%i /= 0)) then
        write (u, "('(',I0,',',I0,')')", advance="no")  object%i
    end if
    write (u, "(A,F7.5,A)")  ": on-shell (", object%m, ")"
end subroutine sf_os_mapping_write
```

Initialize: index pair and dimensionless mass parameter.

⟨*SF mappings: sf on-shell mapping: TBP*⟩+≡
```
    procedure :: init => sf_os_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_os_mapping_init (mapping, m)
      class(sf_os_mapping_t), intent(out) :: mapping
      real(default), intent(in) :: m
      call mapping%base_init (2)
      mapping%m = m
      mapping%lm2 = abs (2 * log (mapping%m))
    end subroutine sf_os_mapping_init
```

Apply mapping. The `x_free` parameter rescales the total energy, which must be accounted for in the enclosed mapping.

⟨*SF mappings: sf on-shell mapping: TBP*⟩+≡
```
    procedure :: compute => sf_os_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_os_mapping_compute (mapping, r, rb, f, p, pb, x_free)
      class(sf_os_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: r2, p2
      integer :: j
      p2 = p(mapping%i)
      call map_on_shell (r2, f, p2, mapping%lm2, x_free)
      r = p
      rb= pb
      do j = 1, 2
         r (mapping%i(j)) = r2(j)
         rb(mapping%i(j)) = 1 - r2(j)
      end do
    end subroutine sf_os_mapping_compute
```

Apply inverse. The irrelevant parameter p_1 is always set zero.

⟨*SF mappings: sf on-shell mapping: TBP*⟩+≡
```
    procedure :: inverse => sf_os_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_os_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_os_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
```

```
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: p2, r2
      r2 = r(mapping%i)
      call map_on_shell_inverse (r2, f, p2, mapping%lm2, x_free)
      p = r
      pb= rb
      p (mapping%i(1)) = p2(1)
      pb(mapping%i(1)) = 1 - p2(1)
      p (mapping%i(2)) = p2(2)
      pb(mapping%i(2)) = 1 - p2(2)
    end subroutine sf_os_mapping_inverse
```

10.2.6   Implementation: endpoint mapping

This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$. The enhancement is such that any power-like singularity is caught. This is useful for beamstrahlung spectra.

In addition, we allow for a delta-function singularity in r_1 and/or r_2. The singularity is smeared to an interval of width ϵ. If nonzero, we distinguish the kinematical momentum fractions r_i from effective values x_i, which should go into the structure-function evaluation. A bin of width ϵ in r is mapped to $x = 1$ exactly, while the interval $(0, 1-\epsilon)$ is mapped to $(0, 1)$ in x. The Jacobian reflects this distinction, and the logical `in_peak` allows for an unambiguous distinction.

The delta-peak fraction is used only for the integration self-test.

⟨*SF mappings: public*⟩+≡
```
    public :: sf_ep_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
    type, extends (sf_mapping_t) :: sf_ep_mapping_t
      real(default) :: a = 1
    contains
    ⟨SF mappings: sf endpoint mapping: TBP⟩
    end type sf_ep_mapping_t
```

Output.

⟨*SF mappings: sf endpoint mapping: TBP*⟩≡
```
    procedure :: write => sf_ep_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ep_mapping_write (object, unit)
      class(sf_ep_mapping_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "map"
      if (any (object%i /= 0)) then
         write (u, "('(',I0,',',I0,')')", advance="no")  object%i
      end if
      write (u, "(A,ES12.5,A)")  ": endpoint (a =", object%a, ")"
    end subroutine sf_ep_mapping_write
```

Initialize: no extra parameters.

⟨SF mappings: sf endpoint mapping: TBP⟩+≡
```
procedure :: init => sf_ep_mapping_init
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_ep_mapping_init (mapping, a)
  class(sf_ep_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: a
  call mapping%base_init (2)
  if (present (a))  mapping%a = a
end subroutine sf_ep_mapping_init
```

Apply mapping.

⟨SF mappings: sf endpoint mapping: TBP⟩+≡
```
procedure :: compute => sf_ep_mapping_compute
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_ep_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_ep_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, r2
  real(default) :: f1, f2
  integer :: j
  call map_endpoint_1 (px(1), f1, p(mapping%i(1)), mapping%a)
  call map_endpoint_01 (px(2), f2, p(mapping%i(2)), mapping%a)
  call map_unit_square (r2, f, px)
  f = f * f1 * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_ep_mapping_compute
```

Apply inverse.

⟨SF mappings: sf endpoint mapping: TBP⟩+≡
```
procedure :: inverse => sf_ep_mapping_inverse
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_ep_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_ep_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, px, p2
  real(default) :: f1, f2
  integer :: j
  do j = 1, 2
```

```
      r2(j) = r(mapping%i(j))
   end do
   call map_unit_square_inverse (r2, f, px)
   call map_endpoint_inverse_1 (px(1), f1, p2(1), mapping%a)
   call map_endpoint_inverse_01 (px(2), f2, p2(2), mapping%a)
   f = f * f1 * f2
   p = r
   pb= rb
   do j = 1, 2
      p (mapping%i(j)) = p2(j)
      pb(mapping%i(j)) = 1 - p2(j)
   end do
end subroutine sf_ep_mapping_inverse
```

10.2.7 Implementation: endpoint mapping with resonance

Like the endpoint mapping for p_2, but replace the endpoint mapping by a Breit-Wigner mapping for p_1. This covers resonance production in the presence of beamstrahlung.

If the flag `resonance` is unset, we skip the resonance mapping, so the parameter p_1 remains equal to $r_1 r_2$, as in the standard s-channel mapping.

⟨*SF mappings: public*⟩+≡
```
   public :: sf_epr_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
   type, extends (sf_mapping_t) :: sf_epr_mapping_t
     real(default) :: a = 1
     real(default) :: m = 0
     real(default) :: w = 0
     logical :: resonance = .true.
    contains
    ⟨SF mappings: sf endpoint/res mapping: TBP⟩
   end type sf_epr_mapping_t
```

Output.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩≡
```
   procedure :: write => sf_epr_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
   subroutine sf_epr_mapping_write (object, unit)
     class(sf_epr_mapping_t), intent(in) :: object
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit)
     write (u, "(1x,A)", advance="no")  "map"
     if (any (object%i /= 0)) then
        write (u, "('(',I0,',',I0,')')", advance="no")  object%i
     end if
     if (object%resonance) then
        write (u, "(A,F7.5,A,F7.5,', ',F7.5,A)")  ": ep/res (a = ", object%a, &
             " | ", object%m, object%w, ")"
     else
```

```
      write (u, "(A,F7.5,A)")  ": ep/nores (a = ", object%a, ")"
   end if
end subroutine sf_epr_mapping_write
```

Initialize: if mass and width are not given, we initialize a non-resonant version of the mapping.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩+≡
```
procedure :: init => sf_epr_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_epr_mapping_init (mapping, a, m, w)
  class(sf_epr_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: a
  real(default), intent(in), optional :: m, w
  call mapping%base_init (2)
  mapping%a = a
  if (present (m) .and. present (w)) then
     mapping%m = m
     mapping%w = w
  else
     mapping%resonance = .false.
  end if
end subroutine sf_epr_mapping_init
```

Apply mapping.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩+≡
```
procedure :: compute => sf_epr_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_epr_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_epr_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, r2
  real(default) :: f1, f2
  integer :: j
  if (mapping%resonance) then
     call map_breit_wigner &
          (px(1), f1, p(mapping%i(1)), mapping%m, mapping%w, x_free)
  else
     px(1) = p(mapping%i(1))
     f1 = 1
  end if
  call map_endpoint_01 (px(2), f2, p(mapping%i(2)), mapping%a)
  call map_unit_square (r2, f, px)
  f = f * f1 * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
```

```
        end subroutine sf_epr_mapping_compute
```

Apply inverse.

⟨SF mappings: sf endpoint/res mapping: TBP⟩+≡
```
    procedure :: inverse => sf_epr_mapping_inverse
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_epr_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_epr_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: px, p2
      real(default) :: f1, f2
      integer :: j
      call map_unit_square_inverse (r(mapping%i), f, px)
      if (mapping%resonance) then
         call map_breit_wigner_inverse &
              (px(1), f1, p2(1), mapping%m, mapping%w, x_free)
      else
         p2(1) = px(1)
         f1 = 1
      end if
      call map_endpoint_inverse_01 (px(2), f2, p2(2), mapping%a)
      f = f * f1 * f2
      p = r
      pb= rb
      do j = 1, 2
         p (mapping%i(j)) = p2(j)
         pb(mapping%i(j)) = 1 - p2(j)
      end do
    end subroutine sf_epr_mapping_inverse
```

10.2.8 Implementation: endpoint mapping for on-shell particle

Analogous to the resonance mapping, but the p_1 input is ignored altogether. This covers on-shell particle production in the presence of beamstrahlung.

⟨SF mappings: public⟩+≡
```
    public :: sf_epo_mapping_t
```

⟨SF mappings: types⟩+≡
```
    type, extends (sf_mapping_t) :: sf_epo_mapping_t
      real(default) :: a = 1
      real(default) :: m = 0
      real(default) :: lm2 = 0
     contains
     ⟨SF mappings: sf endpoint/os mapping: TBP⟩
    end type sf_epo_mapping_t
```

Output.

⟨SF mappings: sf endpoint/os mapping: TBP⟩≡
```
procedure :: write => sf_epo_mapping_write
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_epo_mapping_write (object, unit)
  class(sf_epo_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
      write (u, "('(',I0,',',I0,')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A,F7.5,A)")  ": ep/on-shell (a = ", object%a, &
      " | ", object%m, ")"
end subroutine sf_epo_mapping_write
```

Initialize: no extra parameters.

⟨SF mappings: sf endpoint/os mapping: TBP⟩+≡
```
procedure :: init => sf_epo_mapping_init
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_epo_mapping_init (mapping, a, m)
  class(sf_epo_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: a, m
  call mapping%base_init (2)
  mapping%a = a
  mapping%m = m
  mapping%lm2 = abs (2 * log (mapping%m))
end subroutine sf_epo_mapping_init
```

Apply mapping.

⟨SF mappings: sf endpoint/os mapping: TBP⟩+≡
```
procedure :: compute => sf_epo_mapping_compute
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_epo_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_epo_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, r2
  real(default) :: f2
  integer :: j
  px(1) = 0
  call map_endpoint_01 (px(2), f2, p(mapping%i(2)), mapping%a)
  call map_on_shell (r2, f, px, mapping%lm2)
  f = f * f2
  r = p
  rb= pb
  do j = 1, 2
      r (mapping%i(j)) = r2(j)
```

```
            rb(mapping%i(j)) = 1 - r2(j)
        end do
    end subroutine sf_epo_mapping_compute
```

Apply inverse.

⟨SF mappings: sf endpoint/os mapping: TBP⟩+≡
```
    procedure :: inverse => sf_epo_mapping_inverse
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_epo_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_epo_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: px, p2
      real(default) :: f2
      integer :: j
      call map_on_shell_inverse (r(mapping%i), f, px, mapping%lm2)
      p2(1) = 0
      call map_endpoint_inverse_01 (px(2), f2, p2(2), mapping%a)
      f = f * f2
      p = r
      pb= rb
      do j = 1, 2
          p (mapping%i(j)) = p2(j)
          pb(mapping%i(j)) = 1 - p2(j)
      end do
    end subroutine sf_epo_mapping_inverse
```

10.2.9 Implementation: ISR endpoint mapping

Similar to the endpoint mapping above: This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$.

The enhancement is such that ISR singularity $(1 - x)^{-1+\epsilon}$ is flattened. This would be easy in one dimension, but becomes nontrivial in two dimensions.

⟨SF mappings: public⟩+≡
```
    public :: sf_ip_mapping_t
```

⟨SF mappings: types⟩+≡
```
    type, extends (sf_mapping_t) :: sf_ip_mapping_t
      real(default) :: eps = 0
     contains
```
    ⟨SF mappings: sf power mapping: TBP⟩
```
    end type sf_ip_mapping_t
```

Output.

⟨SF mappings: sf power mapping: TBP⟩≡
```
    procedure :: write => sf_ip_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ip_mapping_write (object, unit)
      class(sf_ip_mapping_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "map"
      if (any (object%i /= 0)) then
          write (u, "('(',I0,',',I0,')')", advance="no")  object%i
      end if
      write (u, "(A,ES12.5,A)")  ": isr (eps =", object%eps, ")"
    end subroutine sf_ip_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf power mapping: TBP*⟩+≡
```
    procedure :: init => sf_ip_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ip_mapping_init (mapping, eps)
      class(sf_ip_mapping_t), intent(out) :: mapping
      real(default), intent(in), optional :: eps
      call mapping%base_init (2)
      if (present (eps))  mapping%eps = eps
      if (mapping%eps <= 0) &
            call msg_fatal ("ISR mapping: regulator epsilon must not be zero")
    end subroutine sf_ip_mapping_init
```

Apply mapping.

⟨*SF mappings: sf power mapping: TBP*⟩+≡
```
    procedure :: compute => sf_ip_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ip_mapping_compute (mapping, r, rb, f, p, pb, x_free)
      class(sf_ip_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: px, pxb, r2, r2b
      real(default) :: f1, f2, xb, y, yb
      integer :: j
      call map_power_1 (xb, f1, pb(mapping%i(1)), 2 * mapping%eps)
      call map_power_01 (y, yb, f2, pb(mapping%i(2)), mapping%eps)
      px(1)  = 1 - xb
      pxb(1) = xb
      px(2)  = y
      pxb(2) = yb
      call map_unit_square_prec (r2, r2b, f, px, pxb)
      f = f * f1 * f2
      r = p
      rb= pb
      do j = 1, 2
          r (mapping%i(j)) = r2 (j)
          rb(mapping%i(j)) = r2b(j)
```

```
        end do
    end subroutine sf_ip_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf power mapping: TBP*⟩+≡
```
    procedure :: inverse => sf_ip_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ip_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_ip_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: r2, r2b, px, pxb, p2, p2b
      real(default) :: f1, f2, xb, y, yb
      integer :: j
      do j = 1, 2
         r2 (j) = r (mapping%i(j))
         r2b(j) = rb(mapping%i(j))
      end do
      call map_unit_square_inverse_prec (r2, r2b, f, px, pxb)
      xb = pxb(1)
      if (px(1) > 0) then
         y  = px(2)
         yb = pxb(2)
      else
         y  = 0.5_default
         yb = 0.5_default
      end if
      call map_power_inverse_1 (xb, f1, p2b(1), 2 * mapping%eps)
      call map_power_inverse_01 (y, yb, f2, p2b(2), mapping%eps)
      p2 = 1 - p2b
      f = f * f1 * f2
      p = r
      pb= rb
      do j = 1, 2
         p (mapping%i(j)) = p2(j)
         pb(mapping%i(j)) = p2b(j)
      end do
    end subroutine sf_ip_mapping_inverse
```

10.2.10 Implementation: ISR endpoint mapping, resonant

Similar to the endpoint mapping above: This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$.

The enhancement is such that ISR singularity $(1-x)^{-1+\epsilon}$ is flattened. This would be easy in one dimension, but becomes nontrivial in two dimensions.

The resonance can be turned off by the flag `resonance`.

⟨*SF mappings: public*⟩+≡

```
      public :: sf_ipr_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
    type, extends (sf_mapping_t) :: sf_ipr_mapping_t
       real(default) :: eps = 0
       real(default) :: m = 0
       real(default) :: w = 0
       logical :: resonance = .true.
     contains
     ⟨SF mappings: sf power/res mapping: TBP⟩
    end type sf_ipr_mapping_t
```

Output.

⟨*SF mappings: sf power/res mapping: TBP*⟩≡
```
    procedure :: write => sf_ipr_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ipr_mapping_write (object, unit)
      class(sf_ipr_mapping_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "map"
      if (any (object%i /= 0)) then
         write (u, "('(',I0,',',I0,')')", advance="no")  object%i
      end if
      if (object%resonance) then
         write (u, "(A,F7.5,A,F7.5,', ',F7.5,A)")  ": isr/res (eps = ", &
              object%eps, " | ", object%m, object%w, ")"
      else
         write (u, "(A,F7.5,A)")  ": isr/res (eps = ", object%eps, ")"
      end if
    end subroutine sf_ipr_mapping_write
```

Initialize:

⟨*SF mappings: sf power/res mapping: TBP*⟩+≡
```
    procedure :: init => sf_ipr_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ipr_mapping_init (mapping, eps, m, w)
      class(sf_ipr_mapping_t), intent(out) :: mapping
      real(default), intent(in), optional :: eps, m, w
      call mapping%base_init (2)
      if (present (eps))  mapping%eps = eps
      if (mapping%eps <= 0) &
           call msg_fatal ("ISR mapping: regulator epsilon must not be zero")
      if (present (m) .and. present (w)) then
         mapping%m = m
         mapping%w = w
      else
         mapping%resonance = .false.
      end if
    end subroutine sf_ipr_mapping_init
```

Apply mapping.

⟨SF mappings: sf power/res mapping: TBP⟩+≡
```
    procedure :: compute => sf_ipr_mapping_compute
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_ipr_mapping_compute (mapping, r, rb, f, p, pb, x_free)
      class(sf_ipr_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: px, pxb, r2, r2b
      real(default) :: f1, f2, y, yb
      integer :: j
      if (mapping%resonance) then
         call map_breit_wigner &
              (px(1), f1, p(mapping%i(1)), mapping%m, mapping%w, x_free)
      else
         px(1) = p(mapping%i(1))
         f1 = 1
      end if
      call map_power_01 (y, yb, f2, pb(mapping%i(2)), mapping%eps)
      pxb(1) = 1 - px(1)
      px(2)  = y
      pxb(2) = yb
      call map_unit_square_prec (r2, r2b, f, px, pxb)
      f = f * f1 * f2
      r = p
      rb= pb
      do j = 1, 2
         r (mapping%i(j)) = r2 (j)
         rb(mapping%i(j)) = r2b(j)
      end do
    end subroutine sf_ipr_mapping_compute
```

Apply inverse.

⟨SF mappings: sf power/res mapping: TBP⟩+≡
```
    procedure :: inverse => sf_ipr_mapping_inverse
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_ipr_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_ipr_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: r2, r2b, px, pxb, p2, p2b
      real(default) :: f1, f2, y, yb
      integer :: j
      do j = 1, 2
         r2 (j) = r (mapping%i(j))
         r2b(j) = rb(mapping%i(j))
      end do
      call map_unit_square_inverse_prec (r2, r2b, f, px, pxb)
```

1030

```
      if (px(1) > 0) then
          y  = px(2)
          yb = pxb(2)
      else
          y  = 0.5_default
          yb = 0.5_default
      end if
      if (mapping%resonance) then
          call map_breit_wigner_inverse &
              (px(1), f1, p2(1), mapping%m, mapping%w, x_free)
      else
          p2(1) = px(1)
          f1 = 1
      end if
      call map_power_inverse_01 (y, yb, f2, p2b(2), mapping%eps)
      p2b(1) = 1 - p2(1)
      p2 (2) = 1 - p2b(2)
      f = f * f1 * f2
      p = r
      pb= rb
      do j = 1, 2
          p (mapping%i(j)) = p2(j)
          pb(mapping%i(j)) = p2b(j)
      end do
  end subroutine sf_ipr_mapping_inverse
```

10.2.11   Implementation: ISR on-shell mapping

Similar to the endpoint mapping above: This maps the unit square (r_1, r_2) such that p_1 is ignored while the product $r_1 r_2$ is constant. p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$.

The enhancement is such that ISR singularity $(1 - x)^{-1+\epsilon}$ is flattened. This would be easy in one dimension, but becomes nontrivial in two dimensions.

⟨*SF mappings: public*⟩+≡
```
    public :: sf_ipo_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
    type, extends (sf_mapping_t) :: sf_ipo_mapping_t
      real(default) :: eps = 0
      real(default) :: m = 0
    contains
    ⟨SF mappings: sf power/os mapping: TBP⟩
    end type sf_ipo_mapping_t
```

Output.
⟨*SF mappings: sf power/os mapping: TBP*⟩≡
```
    procedure :: write => sf_ipo_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ipo_mapping_write (object, unit)
      class(sf_ipo_mapping_t), intent(in) :: object
```

```fortran
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "map"
      if (any (object%i /= 0)) then
          write (u, "('(',I0,',',I0,')')", advance="no")  object%i
      end if
      write (u, "(A,F7.5,A,F7.5,A)")  ": isr/os (eps = ", object%eps, &
          " | ", object%m, ")"
    end subroutine sf_ipo_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf power/os mapping: TBP*⟩+≡
```fortran
    procedure :: init => sf_ipo_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```fortran
    subroutine sf_ipo_mapping_init (mapping, eps, m)
      class(sf_ipo_mapping_t), intent(out) :: mapping
      real(default), intent(in), optional :: eps, m
      call mapping%base_init (2)
      if (present (eps))  mapping%eps = eps
      if (mapping%eps <= 0) &
            call msg_fatal ("ISR mapping: regulator epsilon must not be zero")
      mapping%m = m
    end subroutine sf_ipo_mapping_init
```

Apply mapping.

⟨*SF mappings: sf power/os mapping: TBP*⟩+≡
```fortran
    procedure :: compute => sf_ipo_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```fortran
    subroutine sf_ipo_mapping_compute (mapping, r, rb, f, p, pb, x_free)
      class(sf_ipo_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: px, pxb, r2, r2b
      real(default) :: f1, f2, y, yb
      integer :: j
      call map_power_01 (y, yb, f2, pb(mapping%i(2)), mapping%eps)
      px(1)  = mapping%m ** 2
      if (present (x_free))  px(1) = px(1) / x_free
      pxb(1) = 1 - px(1)
      px(2)  = y
      pxb(2) = yb
      call map_unit_square_prec (r2, r2b, f1, px, pxb)
      f = f1 * f2
      r = p
      rb= pb
      do j = 1, 2
         r (mapping%i(j)) = r2 (j)
         rb(mapping%i(j)) = r2b(j)
      end do
```

```
        end subroutine sf_ipo_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf power/os mapping: TBP*⟩+≡
```
    procedure :: inverse => sf_ipo_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ipo_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_ipo_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: r2, r2b, px, pxb, p2, p2b
      real(default) :: f1, f2, y, yb
      integer :: j
      do j = 1, 2
         r2 (j) = r (mapping%i(j))
         r2b(j) = rb(mapping%i(j))
      end do
      call map_unit_square_inverse_prec (r2, r2b, f1, px, pxb)
      y  = px(2)
      yb = pxb(2)
      call map_power_inverse_01 (y, yb, f2, p2b(2), mapping%eps)
      p2(1) = 0
      p2b(1)= 1
      p2(2) = 1 - p2b(2)
      f = f1 * f2
      p = r
      pb= rb
      do j = 1, 2
         p (mapping%i(j)) = p2(j)
         pb(mapping%i(j)) = p2b(j)
      end do
    end subroutine sf_ipo_mapping_inverse
```

10.2.12 Implementation: Endpoint + ISR power mapping

This is a combination of endpoint (i.e., beamstrahlung) and ISR power mapping.
The first two parameters apply to the beamstrahlung spectrum, the last two to
the ISR function for the first and second beam, respectively.

⟨*SF mappings: public*⟩+≡
```
    public :: sf_ei_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
    type, extends (sf_mapping_t) :: sf_ei_mapping_t
      type(sf_ep_mapping_t) :: ep
      type(sf_ip_mapping_t) :: ip
     contains
```
     ⟨*SF mappings: sf ep-ip mapping: TBP*⟩
```
    end type sf_ei_mapping_t
```

Output.

⟨SF mappings: sf ep-ip mapping: TBP⟩≡
```
    procedure :: write => sf_ei_mapping_write
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_ei_mapping_write (object, unit)
      class(sf_ei_mapping_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "map"
      if (any (object%i /= 0)) then
          write (u, "('(',I0,3(',',I0),')')", advance="no")  object%i
      end if
      write (u, "(A,ES12.5,A,ES12.5,A)")  ": ep/isr (a =", object%ep%a, &
            ", eps =", object%ip%eps, ")"
    end subroutine sf_ei_mapping_write
```

Initialize: no extra parameters.

⟨SF mappings: sf ep-ip mapping: TBP⟩+≡
```
    procedure :: init => sf_ei_mapping_init
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_ei_mapping_init (mapping, a, eps)
      class(sf_ei_mapping_t), intent(out) :: mapping
      real(default), intent(in), optional :: a, eps
      call mapping%base_init (4)
      call mapping%ep%init (a)
      call mapping%ip%init (eps)
    end subroutine sf_ei_mapping_init
```

Set an index value. We should communicate the appropriate indices to the enclosed sub-mappings, therefore override the method.

⟨SF mappings: sf ep-ip mapping: TBP⟩+≡
```
    procedure :: set_index => sf_ei_mapping_set_index
```

⟨SF mappings: procedures⟩+≡
```
    subroutine sf_ei_mapping_set_index (mapping, j, i)
      class(sf_ei_mapping_t), intent(inout) :: mapping
      integer, intent(in) :: j, i
      mapping%i(j) = i
      select case (j)
      case (1:2);  call mapping%ep%set_index (j, i)
      case (3:4);  call mapping%ip%set_index (j-2, i)
      end select
    end subroutine sf_ei_mapping_set_index
```

Apply mapping. Now, the beamstrahlung and ISR mappings are independent of each other. The parameter subsets that are actually used should not overlap. The Jacobians are multiplied.

⟨SF mappings: sf ep-ip mapping: TBP⟩+≡
```
    procedure :: compute => sf_ei_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ei_mapping_compute (mapping, r, rb, f, p, pb, x_free)
      class(sf_ei_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(size(p)) :: q, qb
      real(default) :: f1, f2
      call mapping%ep%compute (q, qb, f1, p, pb, x_free)
      call mapping%ip%compute (r, rb, f2, q, qb, x_free)
      f = f1 * f2
    end subroutine sf_ei_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf ep-ip mapping: TBP*⟩+≡
```
    procedure :: inverse => sf_ei_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_ei_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_ei_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(size(p)) :: q, qb
      real(default) :: f1, f2
      call mapping%ip%inverse (r, rb, f2, q, qb, x_free)
      call mapping%ep%inverse (q, qb, f1, p, pb, x_free)
      f = f1 * f2
    end subroutine sf_ei_mapping_inverse
```

10.2.13   Implementation: Endpoint + ISR + resonance

This is a combination of endpoint (i.e., beamstrahlung) and ISR power mapping, adapted for an s-channel resonance. The first two internal parameters apply to the beamstrahlung spectrum, the last two to the ISR function for the first and second beam, respectively. The first and third parameters are the result of an overall resonance mapping, so on the outside, the first parameter is the total momentum fraction, the third one describes the distribution between beamstrahlung and ISR.

⟨*SF mappings: public*⟩+≡
```
    public :: sf_eir_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
    type, extends (sf_mapping_t) :: sf_eir_mapping_t
      type(sf_res_mapping_t) :: res
      type(sf_epr_mapping_t) :: ep
      type(sf_ipr_mapping_t) :: ip
     contains
```
⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩
```
    end type sf_eir_mapping_t
```

Output.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩≡
```
procedure :: write => sf_eir_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_write (object, unit)
  class(sf_eir_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
      write (u, "('(',I0,3(',',I0),')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A,F7.5,A,F7.5,', ',F7.5,A)")  &
        ": ep/isr/res (a =", object%ep%a, &
        ", eps =", object%ip%eps, " | ", object%res%m, object%res%w, ")"
end subroutine sf_eir_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡
```
procedure :: init => sf_eir_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_init (mapping, a, eps, m, w)
  class(sf_eir_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: a, eps, m, w
  call mapping%base_init (4)
  call mapping%res%init (m, w)
  call mapping%ep%init (a)
  call mapping%ip%init (eps)
end subroutine sf_eir_mapping_init
```

Set an index value. We should communicate the appropriate indices to the
enclosed sub-mappings, therefore override the method.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡
```
procedure :: set_index => sf_eir_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_set_index (mapping, j, i)
  class(sf_eir_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
  select case (j)
  case (1);  call mapping%res%set_index (1, i)
  case (3);  call mapping%res%set_index (2, i)
  end select
  select case (j)
  case (1:2);  call mapping%ep%set_index (j, i)
  case (3:4);  call mapping%ip%set_index (j-2, i)
  end select
end subroutine sf_eir_mapping_set_index
```

Apply mapping. Now, the beamstrahlung and ISR mappings are independent of each other. The parameter subsets that are actually used should not overlap. The Jacobians are multiplied.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡
```
procedure :: compute => sf_eir_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_eir_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(size(p)) :: px, pxb, q, qb
  real(default) :: f0, f1, f2
  call mapping%res%compute (px, pxb, f0, p, pb, x_free)
  call mapping%ep%compute (q, qb, f1, px, pxb, x_free)
  call mapping%ip%compute (r, rb, f2, q, qb, x_free)
  f = f0 * f1 * f2
end subroutine sf_eir_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡
```
procedure :: inverse => sf_eir_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_eir_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(size(p)) :: px, pxb, q, qb
  real(default) :: f0, f1, f2
  call mapping%ip%inverse (r, rb, f2, q, qb, x_free)
  call mapping%ep%inverse (q, qb, f1, px, pxb, x_free)
  call mapping%res%inverse (px, pxb, f0, p, pb, x_free)
  f = f0 * f1 * f2
end subroutine sf_eir_mapping_inverse
```

10.2.14 Implementation: Endpoint + ISR power mapping, on-shell

This is a combination of endpoint (i.e., beamstrahlung) and ISR power mapping. The first two parameters apply to the beamstrahlung spectrum, the last two to the ISR function for the first and second beam, respectively. On top of that, we map the first and third parameter such that the product is constant. From the outside, the first parameter is irrelevant while the third parameter describes the distribution of energy (loss) among beamstrahlung and ISR.

⟨*SF mappings: public*⟩+≡
```
public :: sf_eio_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
type, extends (sf_mapping_t) :: sf_eio_mapping_t
  type(sf_os_mapping_t) :: os
  type(sf_epr_mapping_t) :: ep
  type(sf_ipr_mapping_t) :: ip
 contains
 ⟨SF mappings: sf ep-ip-os mapping: TBP⟩
 end type sf_eio_mapping_t
```

Output.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩≡
```
procedure :: write => sf_eio_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eio_mapping_write (object, unit)
  class(sf_eio_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "('(',I0,3(',',I0),')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A,F7.5,A,F7.5,A)")  ": ep/isr/os (a =", object%ep%a, &
       ", eps =", object%ip%eps, " | ", object%os%m, ")"
end subroutine sf_eio_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡
```
procedure :: init => sf_eio_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eio_mapping_init (mapping, a, eps, m)
  class(sf_eio_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: a, eps, m
  call mapping%base_init (4)
  call mapping%os%init (m)
  call mapping%ep%init (a)
  call mapping%ip%init (eps)
end subroutine sf_eio_mapping_init
```

Set an index value. We should communicate the appropriate indices to the enclosed sub-mappings, therefore override the method.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡
```
procedure :: set_index => sf_eio_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eio_mapping_set_index (mapping, j, i)
  class(sf_eio_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
  select case (j)
  case (1);  call mapping%os%set_index (1, i)
```

```
      case (3);  call mapping%os%set_index (2, i)
      end select
      select case (j)
      case (1:2);  call mapping%ep%set_index (j, i)
      case (3:4);  call mapping%ip%set_index (j-2, i)
      end select
    end subroutine sf_eio_mapping_set_index
```

Apply mapping. Now, the beamstrahlung and ISR mappings are independent of each other. The parameter subsets that are actually used should not overlap. The Jacobians are multiplied.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡
```
    procedure :: compute => sf_eio_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_eio_mapping_compute (mapping, r, rb, f, p, pb, x_free)
      class(sf_eio_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(size(p)) :: px, pxb, q, qb
      real(default) :: f0, f1, f2
      call mapping%os%compute (px, pxb, f0, p, pb, x_free)
      call mapping%ep%compute (q, qb, f1, px, pxb, x_free)
      call mapping%ip%compute (r, rb, f2, q, qb, x_free)
      f = f0 * f1 * f2
    end subroutine sf_eio_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡
```
    procedure :: inverse => sf_eio_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_eio_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
      class(sf_eio_mapping_t), intent(inout) :: mapping
      real(default), dimension(:), intent(in) :: r, rb
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
      real(default), dimension(size(p)) :: px, pxb, q, qb
      real(default) :: f0, f1, f2
      call mapping%ip%inverse (r, rb, f2, q, qb, x_free)
      call mapping%ep%inverse (q, qb, f1, px, pxb, x_free)
      call mapping%os%inverse (px, pxb, f0, p, pb, x_free)
      f = f0 * f1 * f2
    end subroutine sf_eio_mapping_inverse
```

10.2.15 Basic formulas

Standard mapping of the unit square

This mapping of the unit square is appropriate in particular for structure functions which are concentrated at the lower end. Instead of a rectangular grid, one set of grid lines corresponds to constant parton c.m. energy. The other set is chosen such that the jacobian is only mildly singular ($\ln x$ which is zero at $x = 1$), corresponding to an initial concentration of sampling points at the maximum energy. If `power` is greater than one (the default), points are also concentrated at the lower end.

The formula is (`power`$=\alpha$):

$$r_1 = (p_1^{p_2})^\alpha \tag{10.15}$$

$$r_2 = (p_1^{1-p_2})^\alpha \tag{10.16}$$

$$f = \alpha^2 p_1^{\alpha-1} |\log p_1| \tag{10.17}$$

and for the default case $\alpha = 1$:

$$r_1 = p_1^{p_2} \tag{10.18}$$

$$r_2 = p_1^{1-p_2} \tag{10.19}$$

$$f = |\log p_1| \tag{10.20}$$

⟨*SF mappings: procedures*⟩+≡
```
subroutine map_unit_square (r, factor, p, power)
  real(default), dimension(2), intent(out) :: r
  real(default), intent(out) :: factor
  real(default), dimension(2), intent(in) :: p
  real(default), intent(in), optional :: power
  real(default) :: xx, yy
  factor = 1
  xx = p(1)
  yy = p(2)
  if (present(power)) then
     if (p(1) > 0 .and. power > 1) then
        xx = p(1)**power
        factor = factor * power * xx / p(1)
     end if
  end if
  if (xx /= 0) then
     r(1) = xx ** yy
     r(2) = xx / r(1)
     factor = factor * abs (log (xx))
  else
     r = 0
  end if
end subroutine map_unit_square
```

This is the inverse mapping.

⟨*SF mappings: procedures*⟩+≡
```
subroutine map_unit_square_inverse (r, factor, p, power)
  real(kind=default), dimension(2), intent(in) :: r
```

```
      real(kind=default), intent(out) :: factor
      real(kind=default), dimension(2), intent(out) :: p
      real(kind=default), intent(in), optional :: power
      real(kind=default) :: lg, xx, yy
      factor = 1
      xx = r(1) * r(2)
      if (xx /= 0) then
         lg = log (xx)
         if (lg /= 0) then
            yy = log (r(1)) / lg
         else
            yy = 0
         end if
         p(2) = yy
         factor = factor * abs (lg)
         if (present(power)) then
            p(1) = xx**(1._default/power)
            factor = factor * power * xx / p(1)
         else
            p(1) = xx
         end if
      else
         p = 0
      end if
   end subroutine map_unit_square_inverse
```

Precise mapping of the unit square

A more precise version (with unit power parameter). This version should be numerically stable near $x = 1$ and $y = 0, 1$. The formulas are again

$$r_1 = p_1^{p_2}, \qquad r_2 = p_1^{\bar{p}_2}, \qquad f = -\log p_1 \qquad (10.21)$$

but we compute both r_i and $\bar{r}_i$ simultaneously and make direct use of both p_i and $\bar{p}_i$ as appropriate.

⟨*SF mappings: procedures*⟩+≡
```
   subroutine map_unit_square_prec (r, rb, factor, p, pb)
     real(default), dimension(2), intent(out) :: r
     real(default), dimension(2), intent(out) :: rb
     real(default), intent(out) :: factor
     real(default), dimension(2), intent(in) :: p
     real(default), dimension(2), intent(in) :: pb
     if (p(1) > 0.5_default) then
        call compute_prec_xy_1 (r(1), rb(1), p(1), pb(1), p (2))
        call compute_prec_xy_1 (r(2), rb(2), p(1), pb(1), pb(2))
        factor = - log_prec (p(1), pb(1))
     else if (p(1) > 0) then
        call compute_prec_xy_0 (r(1), rb(1), p(1), pb(1), p (2))
        call compute_prec_xy_0 (r(2), rb(2), p(1), pb(1), pb(2))
        factor = - log_prec (p(1), pb(1))
     else
        r  = 0
        rb = 1
```

```
        factor = 0
      end if
    end subroutine map_unit_square_prec
```

This is the inverse mapping.

⟨*SF mappings: procedures*⟩+≡
```
    subroutine map_unit_square_inverse_prec (r, rb, factor, p, pb)
      real(default), dimension(2), intent(in) :: r
      real(default), dimension(2), intent(in) :: rb
      real(default), intent(out) :: factor
      real(default), dimension(2), intent(out) :: p
      real(default), dimension(2), intent(out) :: pb
      call inverse_prec_x (r, rb, p(1), pb(1))
      if (all (r > 0)) then
         if (rb(1) < rb(2)) then
            call inverse_prec_y (r, rb, p(2), pb(2))
         else
            call inverse_prec_y ([r(2),r(1)], [rb(2),rb(1)], pb(2), p(2))
         end if
         factor = - log_prec (p(1), pb(1))
      else
         p(1)  = 0
         pb(1) = 1
         p(2)  = 0.5_default
         pb(2) = 0.5_default
         factor = 0
      end if
    end subroutine map_unit_square_inverse_prec
```

This is an auxiliary function: evaluate the expression $\bar{z} = 1 - x^y$ in a numerically stable way. Instabilities occur for $y = 0$ and $x = 1$. The idea is to replace the bracket by the first terms of its Taylor expansion around $x = 1$ (read $\bar{x} \equiv 1 - x$)

$$1 - x^y = y\bar{x}\left(1 + \frac{1}{2}(1 - y)\bar{x} + \frac{1}{6}(2 - y)(1 - y)\bar{x}^2\right) \qquad (10.22)$$

whenever this is the better approximation. Actually, the relative numerical error of the exact formula is about $\eta/(y\bar{x})$ where η is given by `epsilon(KIND)` in Fortran. The relative error of the approximation is better than the last included term divided by $(y\bar{x})$.

The first subroutine computes z and $\bar{z}$ near $x = 1$ where $\log x$ should be expanded, the second one near $x = 0$ where $\log x$ can be kept.

⟨*SF mappings: procedures*⟩+≡
```
    subroutine compute_prec_xy_1 (z, zb, x, xb, y)
      real(default), intent(out) :: z, zb
      real(default), intent(in) :: x, xb, y
      real(default) :: a1, a2, a3
      a1 = y * xb
      a2 = a1 * (1 - y) * xb / 2
      a3 = a2 * (2 - y) * xb / 3
      if (abs (a3) < epsilon (a3)) then
         zb = a1 + a2 + a3
         z = 1 - zb
```

```
    else
        z = x ** y
        zb = 1 - z
    end if
end subroutine compute_prec_xy_1

subroutine compute_prec_xy_0 (z, zb, x, xb, y)
    real(default), intent(out) :: z, zb
    real(default), intent(in) :: x, xb, y
    real(default) :: a1, a2, a3, lx
    lx = -log (x)
    a1 = y * lx
    a2 = a1 * y * lx / 2
    a3 = a2 * y * lx / 3
    if (abs (a3) < epsilon (a3)) then
        zb = a1 + a2 + a3
        z = 1 - zb
    else
        z = x ** y
        zb = 1 - z
    end if
end subroutine compute_prec_xy_0
```

For the inverse calculation, we evaluate $x = r_1 r_2$ in a stable way. Since it is just a polynomial, the expansion near $x = 1$ is analytically exact, and we don't need to choose based on precision.

⟨*SF mappings: procedures*⟩+≡
```
subroutine inverse_prec_x (r, rb, x, xb)
    real(default), dimension(2), intent(in) :: r, rb
    real(default), intent(out) :: x, xb
    real(default) :: a0, a1
    a0 = rb(1) + rb(2)
    a1 = rb(1) * rb(2)
    if (a0 > 0.5_default) then
        xb = a0 - a1
        x = 1 - xb
    else
        x = r(1) * r(2)
        xb = 1 - x
    end if
end subroutine inverse_prec_x
```

The inverse calculation for the relative momentum fraction

$$y = \frac{1}{1 + \frac{\log r_2}{\log r_1}} \tag{10.23}$$

is slightly more complicated. We should take the precise form of the logarithm, so we are safe near $r_i = 1$. A series expansion is required if $r_1 \ll r_2$, since then y becomes small. (We assume $r_1 < r_2$ here; for the opposite case, the arguments can be exchanged.)

⟨*SF mappings: procedures*⟩+≡
```
subroutine inverse_prec_y (r, rb, y, yb)
```

```fortran
    real(default), dimension(2), intent(in) :: r, rb
    real(default), intent(out) :: y, yb
    real(default) :: log1, log2, a1, a2, a3
    log1 = log_prec (r(1), rb(1))
    log2 = log_prec (r(2), rb(2))
    a1 = - rb(1) / log2
    a2 = - rb(1) ** 2 * (1 / log2**2 + 1 / (2 * log2))
    a3 = - rb(1) ** 3 * (1 / log2**3 + 1 / log2**2 + 1 / (3 * log2))
    if (abs (a3) < epsilon (a3)) then
       y  = a1 + a2 + a3
       yb = 1 - y
    else
       y  = 1 / (1 + log2/log1)
       yb = 1 / (1 + log1/log2)
    end if
  end subroutine inverse_prec_y
```

We also need an evaluation of $\log x$ which is stable near $x = 1$.

⟨*SF mappings: public*⟩+≡
```fortran
    public :: log_prec
```

⟨*SF mappings: procedures*⟩+≡
```fortran
    function log_prec (x, xb) result (lx)
      real(default), intent(in) :: x, xb
      real(default) :: a1, a2, a3, lx
      a1 = xb
      a2 = a1 * xb / 2
      a3 = a2 * xb * 2 / 3
      if (abs (a3) < epsilon (a3)) then
         lx = - a1 - a2 - a3
      else
         lx = log (x)
      end if
    end function log_prec
```

Mapping for on-shell s-channel

The limiting case, if the product $r_1 r_2$ is fixed for on-shell production. The parameter p_1 is ignored. In the inverse mapping, it is returned zero.

The parameter `x_free`, if present, rescales the total energy. If it is less than one, the rescaled mass parameter m^2 should be increased accordingly.

⟨*SF mappings: procedures*⟩+≡
```fortran
    subroutine map_on_shell (r, factor, p, lm2, x_free)
      real(default), dimension(2), intent(out) :: r
      real(default), intent(out) :: factor
      real(default), dimension(2), intent(in) :: p
      real(default), intent(in) :: lm2
      real(default), intent(in), optional :: x_free
      real(default) :: lx
      lx = lm2;  if (present (x_free))  lx = lx + log (x_free)
      r(1) = exp (- p(2) * lx)
      r(2) = exp (- (1 - p(2)) * lx)
```

```
      factor = lx
    end subroutine map_on_shell

    subroutine map_on_shell_inverse (r, factor, p, lm2, x_free)
      real(default), dimension(2), intent(in) :: r
      real(default), intent(out) :: factor
      real(default), dimension(2), intent(out) :: p
      real(default), intent(in) :: lm2
      real(default), intent(in), optional :: x_free
      real(default) :: lx
      lx = lm2;  if (present (x_free))  lx = lx + log (x_free)
      p(1) = 0
      p(2) = abs (log (r(1))) / lx
      factor = lx
    end subroutine map_on_shell_inverse
```

This is the standard Breit-Wigner mapping. We apply it to a single variable, independently of or in addition to a unit-square mapping. We assume here that the limits for the variable are 0 and 1, and that the mass m and width w are rescaled appropriately, so they are dimensionless and usually between 0 and 1.

If `x_free` is set, it rescales the total energy and thus mass and width, since these are defined with respect to the total energy.

⟨*SF mappings: procedures*⟩+≡

```
    subroutine map_breit_wigner (r, factor, p, m, w, x_free)
      real(default), intent(out) :: r
      real(default), intent(out) :: factor
      real(default), intent(in) :: p
      real(default), intent(in) :: m
      real(default), intent(in) :: w
      real(default), intent(in), optional :: x_free
      real(default) :: m2, mw, a1, a2, a3, z, tmp
      m2 = m ** 2
      mw = m * w
      if (present (x_free)) then
         m2 = m2 / x_free
         mw = mw / x_free
      end if
      a1 = atan (- m2 / mw)
      a2 = atan ((1 - m2) / mw)
      a3 = (a2 - a1) * mw
      z = (1-p) * a1 + p * a2
      if (-pi/2 < z .and. z < pi/2) then
         tmp = tan (z)
         r = max (m2 + mw * tmp, 0._default)
         factor = a3 * (1 + tmp ** 2)
      else
         r = 0
         factor = 0
      end if
    end subroutine map_breit_wigner

    subroutine map_breit_wigner_inverse (r, factor, p, m, w, x_free)
      real(default), intent(in) :: r
```

```
      real(default), intent(out) :: factor
      real(default), intent(out) :: p
      real(default), intent(in) :: m
      real(default), intent(in) :: w
      real(default) :: m2, mw, a1, a2, a3, tmp
      real(default), intent(in), optional :: x_free
      m2 = m ** 2
      mw = m * w
      if (present (x_free)) then
         m2 = m2 / x_free
         mw = mw / x_free
      end if
      a1 = atan (- m2 / mw)
      a2 = atan ((1 - m2) / mw)
      a3 = (a2 - a1) * mw
      tmp = (r - m2) / mw
      p = (atan (tmp) - a1) / (a2 - a1)
      factor = a3 * (1 + tmp ** 2)
    end subroutine map_breit_wigner_inverse
```

Mapping with endpoint enhancement

This is a mapping which is close to the unit mapping, except that at the end-point(s), the output values are exponentially enhanced.

$$y = \tanh(a \tan(\frac{\pi}{2}x)) \tag{10.24}$$

We have two variants: one covers endpoints at 0 and 1 symmetrically, while the other one (which essentially maps one-half of the range), covers only the endpoint at 1.

⟨*SF mappings: procedures*⟩+≡

```
    subroutine map_endpoint_1 (x3, factor, x1, a)
      real(default), intent(out) :: x3, factor
      real(default), intent(in) :: x1
      real(default), intent(in) :: a
      real(default) :: x2
      if (abs (x1) < 1) then
         x2 = tan (x1 * pi / 2)
         x3 = tanh (a * x2)
         factor = a * pi/2 * (1 + x2 ** 2) * (1 - x3 ** 2)
      else
         x3 = x1
         factor = 0
      end if
    end subroutine map_endpoint_1

    subroutine map_endpoint_inverse_1 (x3, factor, x1, a)
      real(default), intent(in) :: x3
      real(default), intent(out) :: x1, factor
      real(default), intent(in) :: a
      real(default) :: x2
      if (abs (x3) < 1) then
```

```
          x2 = atanh (x3) / a
          x1 = 2 / pi * atan (x2)
          factor = a * pi/2 * (1 + x2 ** 2) * (1 - x3 ** 2)
       else
          x1 = x3
          factor = 0
       end if
    end subroutine map_endpoint_inverse_1

    subroutine map_endpoint_01 (x4, factor, x0, a)
      real(default), intent(out) :: x4, factor
      real(default), intent(in) :: x0
      real(default), intent(in) :: a
      real(default) :: x1, x3
      x1 = 2 * x0 - 1
      call map_endpoint_1 (x3, factor, x1, a)
      x4 = (x3 + 1) / 2
    end subroutine map_endpoint_01

    subroutine map_endpoint_inverse_01 (x4, factor, x0, a)
      real(default), intent(in) :: x4
      real(default), intent(out) :: x0, factor
      real(default), intent(in) :: a
      real(default) :: x1, x3
      x3 = 2 * x4 - 1
      call map_endpoint_inverse_1 (x3, factor, x1, a)
      x0 = (x1 + 1) / 2
    end subroutine map_endpoint_inverse_01
```

Mapping with endpoint enhancement (ISR)

This is another endpoint mapping. It is designed to flatten the ISR singularity
which is of power type at $x = 1$, i.e., if

$$\sigma = \int_0^1 dx\, f(x)\, G(x) = \int_0^1 dx\, \epsilon (1 - x)^{-1+\epsilon} G(x), \qquad (10.25)$$

we replace this by

$$r = x^\epsilon \quad \Longrightarrow \quad \sigma = \int_0^1 dr\, G(1 - (1 - r)^{1/\epsilon}). \qquad (10.26)$$

We expect that ϵ is small.

The actual mapping is $r \to x$ (so x emerges closer to 1). The Jacobian that
we return is thus $1/f(x)$. We compute the mapping in terms of $\bar{x} \equiv 1 - x$, so
we can achieve the required precision. Because some compilers show quite wild
numeric fluctuations, we internally convert numeric types to explicit `double`
precision.

⟨*SF mappings: public*⟩+≡
```
    public :: map_power_1
    public :: map_power_inverse_1
```

⟨*SF mappings: procedures*⟩+≡

```fortran
    subroutine map_power_1 (xb, factor, rb, eps)
      real(default), intent(out) :: xb, factor
      real(default), intent(in) :: rb
      real(double) :: rb_db, factor_db, eps_db, xb_db
      real(default), intent(in) :: eps
      rb_db = real (rb, kind=double)
      eps_db = real (eps, kind=double)
      xb_db = rb_db ** (1 / eps_db)
      if (rb_db > 0) then
         factor_db = xb_db / rb_db / eps_db
         factor = real (factor_db, kind=default)
      else
         factor = 0
      end if
      xb = real (xb_db, kind=default)
    end subroutine map_power_1

    subroutine map_power_inverse_1 (xb, factor, rb, eps)
      real(default), intent(in) :: xb
      real(default), intent(out) :: rb, factor
      real(double) :: xb_db, factor_db, eps_db, rb_db
      real(default), intent(in) :: eps
      xb_db = real (xb, kind=double)
      eps_db = real (eps, kind=double)
      rb_db = xb_db ** eps_db
      if (xb_db > 0) then
         factor_db = xb_db / rb_db / eps_db
         factor = real (factor_db, kind=default)
      else
         factor = 0
      end if
      rb = real (rb_db, kind=default)
    end subroutine map_power_inverse_1
```

Here we apply a power mapping to both endpoints. We divide the interval in two equal halves and apply the power mapping for the nearest endpoint, either 0 or 1.

⟨*SF mappings: procedures*⟩+≡

```fortran
    subroutine map_power_01 (y, yb, factor, r, eps)
      real(default), intent(out) :: y, yb, factor
      real(default), intent(in) :: r
      real(default), intent(in) :: eps
      real(default) :: u, ub, zp, zm
      u = 2 * r - 1
      if (u > 0) then
         ub = 2 * (1 - r)
         call map_power_1 (zm, factor, ub, eps)
         zp = 2 - zm
      else if (u < 0) then
         ub = 2 * r
         call map_power_1 (zp, factor, ub, eps)
         zm = 2 - zp
```

1048

```
    else
       factor = 1 / eps
       zp = 1
       zm = 1
    end if
    y  = zp / 2
    yb = zm / 2
  end subroutine map_power_01

  subroutine map_power_inverse_01 (y, yb, factor, r, eps)
    real(default), intent(in) :: y, yb
    real(default), intent(out) :: r, factor
    real(default), intent(in) :: eps
    real(default) :: ub, zp, zm
    zp = 2 * y
    zm = 2 * yb
    if (zm < zp) then
       call map_power_inverse_1 (zm, factor, ub, eps)
       r = 1 - ub / 2
    else if (zp < zm) then
       call map_power_inverse_1 (zp, factor, ub, eps)
       r = ub / 2
    else
       factor = 1 / eps
       ub = 1
       r = ub / 2
    end if
  end subroutine map_power_inverse_01
```

Structure-function channels

A structure-function chain parameterization (channel) may contain a mapping
that applies to multiple structure functions. This is described by an extension
of the `sf_mapping_t` type. In addition, it may contain mappings that apply
to (other) individual structure functions. The details of these mappings are
implementation-specific.

The `sf_channel_t` type combines this information. It contains an array of
map codes, one for each structure-function entry. The code values are:

none MC input parameters r directly become energy fractions x

single default mapping for a single structure-function entry

multi/s map $r \to x$ such that one MC input parameter is $\hat{s}/s$

multi/resonance as before, adapted to s-channel resonance

multi/on-shell as before, adapted to an on-shell particle in the s channel

multi/endpoint like multi/s, but enhance the region near $r_i = 1$

multi/endpoint/res endpoint mapping with resonance

multi/endpoint/os endpoint mapping for on-shell

1049

multi/power/os like multi/endpoint, regulating a power singularity

⟨*SF mappings: parameters*⟩≡
```
integer, parameter :: SFMAP_NONE = 0
integer, parameter :: SFMAP_SINGLE = 1
integer, parameter :: SFMAP_MULTI_S = 2
integer, parameter :: SFMAP_MULTI_RES = 3
integer, parameter :: SFMAP_MULTI_ONS = 4
integer, parameter :: SFMAP_MULTI_EP = 5
integer, parameter :: SFMAP_MULTI_EPR = 6
integer, parameter :: SFMAP_MULTI_EPO = 7
integer, parameter :: SFMAP_MULTI_IP = 8
integer, parameter :: SFMAP_MULTI_IPR = 9
integer, parameter :: SFMAP_MULTI_IPO = 10
integer, parameter :: SFMAP_MULTI_EI = 11
```

Then, it contains an allocatable entry for the multi mapping. This entry holds the MC-parameter indices on which the mapping applies (there may be more than one MC parameter per structure-function entry) and any parameters associated with the mapping.

There can be only one multi-mapping per channel.

⟨*SF mappings: public*⟩+≡
```
public :: sf_channel_t
```

⟨*SF mappings: types*⟩+≡
```
type :: sf_channel_t
   integer, dimension(:), allocatable :: map_code
   class(sf_mapping_t), allocatable :: multi_mapping
 contains
 ⟨SF mappings: sf channel: TBP⟩
 end type sf_channel_t
```

The output format prints a single character for each structure-function entry and, if applicable, an account of the mapping parameters.

⟨*SF mappings: sf channel: TBP*⟩≡
```
procedure :: write => sf_channel_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_channel_write (object, unit)
  class(sf_channel_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit)
  if (allocated (object%map_code)) then
    do i = 1, size (object%map_code)
       select case (object%map_code (i))
       case (SFMAP_NONE)
          write (u, "(1x,A)", advance="no") "-"
       case (SFMAP_SINGLE)
          write (u, "(1x,A)", advance="no") "+"
       case (SFMAP_MULTI_S)
          write (u, "(1x,A)", advance="no") "s"
       case (SFMAP_MULTI_RES)
          write (u, "(1x,A)", advance="no") "r"
```

```
            case (SFMAP_MULTI_ONS)
               write (u, "(1x,A)", advance="no") "o"
            case (SFMAP_MULTI_EP)
               write (u, "(1x,A)", advance="no") "e"
            case (SFMAP_MULTI_EPR)
               write (u, "(1x,A)", advance="no") "p"
            case (SFMAP_MULTI_EPO)
               write (u, "(1x,A)", advance="no") "q"
            case (SFMAP_MULTI_IP)
               write (u, "(1x,A)", advance="no") "i"
            case (SFMAP_MULTI_IPR)
               write (u, "(1x,A)", advance="no") "i"
            case (SFMAP_MULTI_IPO)
               write (u, "(1x,A)", advance="no") "i"
            case (SFMAP_MULTI_EI)
               write (u, "(1x,A)", advance="no") "i"
            end select
         end do
      else
         write (u, "(1x,A)", advance="no") "-"
      end if
      if (allocated (object%multi_mapping)) then
         write (u, "(1x,'/')", advance="no")
         call object%multi_mapping%write (u)
      else
         write (u, *)
      end if
    end subroutine sf_channel_write
```

Initializer for a single `sf_channel` object.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: init => sf_channel_init
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_init (channel, n_strfun)
      class(sf_channel_t), intent(out) :: channel
      integer, intent(in) :: n_strfun
      allocate (channel%map_code (n_strfun))
      channel%map_code = SFMAP_NONE
    end subroutine sf_channel_init
```

Assignment. This merely copies intrinsic assignment, but apparently the latter is bugged in gfortran 4.6.3, causing memory corruption.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    generic :: assignment (=) => sf_channel_assign
    procedure :: sf_channel_assign
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_assign (copy, original)
      class(sf_channel_t), intent(out) :: copy
      type(sf_channel_t), intent(in) :: original
      allocate (copy%map_code (size (original%map_code)))
      copy%map_code = original%map_code
      if (allocated (original%multi_mapping)) then
```

```
          allocate (copy%multi_mapping, source = original%multi_mapping)
        end if
      end subroutine sf_channel_assign
```

This initializer allocates an array of channels with common number of structure-function entries, therefore it is not a type-bound procedure.

⟨*SF mappings: public*⟩+≡
```
      public :: allocate_sf_channels
```

⟨*SF mappings: procedures*⟩+≡
```
      subroutine allocate_sf_channels (channel, n_channel, n_strfun)
        type(sf_channel_t), dimension(:), intent(out), allocatable :: channel
        integer, intent(in) :: n_channel
        integer, intent(in) :: n_strfun
        integer :: c
        allocate (channel (n_channel))
        do c = 1, n_channel
           call channel(c)%init (n_strfun)
        end do
      end subroutine allocate_sf_channels
```

This marks a given subset of indices as single-mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
      procedure :: activate_mapping => sf_channel_activate_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
      subroutine sf_channel_activate_mapping (channel, i_sf)
        class(sf_channel_t), intent(inout) :: channel
        integer, dimension(:), intent(in) :: i_sf
        channel%map_code(i_sf) = SFMAP_SINGLE
      end subroutine sf_channel_activate_mapping
```

This sets an s-channel multichannel mapping. The length of the `i_sf` array must be 2. The parameter indices are not yet set.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
      procedure :: set_s_mapping => sf_channel_set_s_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
      subroutine sf_channel_set_s_mapping (channel, i_sf, power)
        class(sf_channel_t), intent(inout) :: channel
        integer, dimension(:), intent(in) :: i_sf
        real(default), intent(in), optional :: power
        channel%map_code(i_sf) = SFMAP_MULTI_S
        allocate (sf_s_mapping_t :: channel%multi_mapping)
        select type (mapping => channel%multi_mapping)
        type is (sf_s_mapping_t)
           call mapping%init (power)
        end select
      end subroutine sf_channel_set_s_mapping
```

This sets an s-channel resonance multichannel mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
      procedure :: set_res_mapping => sf_channel_set_res_mapping
```

1052

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_res_mapping (channel, i_sf, m, w)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in) :: m, w
      channel%map_code(i_sf) = SFMAP_MULTI_RES
      allocate (sf_res_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_res_mapping_t)
         call mapping%init (m, w)
      end select
    end subroutine sf_channel_set_res_mapping
```

This sets an s-channel on-shell multichannel mapping. The length of the `i_sf`
array must be 2. (The first parameter actually becomes an irrelevant dummy.)

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_os_mapping => sf_channel_set_os_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_os_mapping (channel, i_sf, m)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in) :: m
      channel%map_code(i_sf) = SFMAP_MULTI_ONS
      allocate (sf_os_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_os_mapping_t)
         call mapping%init (m)
      end select
    end subroutine sf_channel_set_os_mapping
```

This sets an s-channel endpoint mapping. The parameter a is the slope parameter (default 1); increasing it moves the endpoint region (at $x = 1$ to lower values in the input parameter. region even more.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_ep_mapping => sf_channel_set_ep_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_ep_mapping (channel, i_sf, a)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in), optional :: a
      channel%map_code(i_sf) = SFMAP_MULTI_EP
      allocate (sf_ep_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_ep_mapping_t)
         call mapping%init (a = a)
      end select
    end subroutine sf_channel_set_ep_mapping
```

This sets a resonant endpoint mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_epr_mapping => sf_channel_set_epr_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_epr_mapping (channel, i_sf, a, m, w)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in) :: a, m, w
      channel%map_code(i_sf) = SFMAP_MULTI_EPR
      allocate (sf_epr_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_epr_mapping_t)
         call mapping%init (a, m, w)
      end select
    end subroutine sf_channel_set_epr_mapping
```

This sets an on-shell endpoint mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_epo_mapping => sf_channel_set_epo_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_epo_mapping (channel, i_sf, a, m)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in) :: a, m
      channel%map_code(i_sf) = SFMAP_MULTI_EPO
      allocate (sf_epo_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_epo_mapping_t)
         call mapping%init (a, m)
      end select
    end subroutine sf_channel_set_epo_mapping
```

This sets an s-channel power mapping, regulating a singularity of type $(1 - x)^{-1+\epsilon}$. The parameter ϵ depends on the structure function.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_ip_mapping => sf_channel_set_ip_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_ip_mapping (channel, i_sf, eps)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in), optional :: eps
      channel%map_code(i_sf) = SFMAP_MULTI_IP
      allocate (sf_ip_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_ip_mapping_t)
         call mapping%init (eps)
      end select
    end subroutine sf_channel_set_ip_mapping
```

This sets an s-channel resonant power mapping, regulating a singularity of type $(1-x)^{-1+\epsilon}$ in the presence of an s-channel resonance. The parameter ϵ depends on the structure function.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_ipr_mapping => sf_channel_set_ipr_mapping
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_channel_set_ipr_mapping (channel, i_sf, eps, m, w)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  real(default), intent(in), optional :: eps, m, w
  channel%map_code(i_sf) = SFMAP_MULTI_IPR
  allocate (sf_ipr_mapping_t :: channel%multi_mapping)
  select type (mapping => channel%multi_mapping)
  type is (sf_ipr_mapping_t)
     call mapping%init (eps, m, w)
  end select
end subroutine sf_channel_set_ipr_mapping
```

This sets an on-shell power mapping, regulating a singularity of type $(1-x)^{-1+\epsilon}$ for the production of a single on-shell particle.. The parameter ϵ depends on the structure function.

⟨SF mappings: sf channel: TBP⟩+≡
```
procedure :: set_ipo_mapping => sf_channel_set_ipo_mapping
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_channel_set_ipo_mapping (channel, i_sf, eps, m)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  real(default), intent(in), optional :: eps, m
  channel%map_code(i_sf) = SFMAP_MULTI_IPO
  allocate (sf_ipo_mapping_t :: channel%multi_mapping)
  select type (mapping => channel%multi_mapping)
  type is (sf_ipo_mapping_t)
     call mapping%init (eps, m)
  end select
end subroutine sf_channel_set_ipo_mapping
```

This sets a combined endpoint/ISR mapping.

⟨SF mappings: sf channel: TBP⟩+≡
```
procedure :: set_ei_mapping => sf_channel_set_ei_mapping
```

⟨SF mappings: procedures⟩+≡
```
subroutine sf_channel_set_ei_mapping (channel, i_sf, a, eps)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  real(default), intent(in), optional :: a, eps
  channel%map_code(i_sf) = SFMAP_MULTI_EI
  allocate (sf_ei_mapping_t :: channel%multi_mapping)
  select type (mapping => channel%multi_mapping)
  type is (sf_ei_mapping_t)
     call mapping%init (a, eps)
  end select
end subroutine sf_channel_set_ei_mapping
```

This sets a combined endpoint/ISR mapping with resonance.

⟨SF mappings: sf channel: TBP⟩+≡
```
procedure :: set_eir_mapping => sf_channel_set_eir_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_eir_mapping (channel, i_sf, a, eps, m, w)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in), optional :: a, eps, m, w
      channel%map_code(i_sf) = SFMAP_MULTI_EI
      allocate (sf_eir_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_eir_mapping_t)
         call mapping%init (a, eps, m, w)
      end select
    end subroutine sf_channel_set_eir_mapping
```

This sets a combined endpoint/ISR mapping, on-shell.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: set_eio_mapping => sf_channel_set_eio_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    subroutine sf_channel_set_eio_mapping (channel, i_sf, a, eps, m)
      class(sf_channel_t), intent(inout) :: channel
      integer, dimension(:), intent(in) :: i_sf
      real(default), intent(in), optional :: a, eps, m
      channel%map_code(i_sf) = SFMAP_MULTI_EI
      allocate (sf_eio_mapping_t :: channel%multi_mapping)
      select type (mapping => channel%multi_mapping)
      type is (sf_eio_mapping_t)
         call mapping%init (a, eps, m)
      end select
    end subroutine sf_channel_set_eio_mapping
```

Return true if the mapping code at position `i_sf` is `SFMAP_SINGLE`.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: is_single_mapping => sf_channel_is_single_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    function sf_channel_is_single_mapping (channel, i_sf) result (flag)
      class(sf_channel_t), intent(in) :: channel
      integer, intent(in) :: i_sf
      logical :: flag
      flag = channel%map_code(i_sf) == SFMAP_SINGLE
    end function sf_channel_is_single_mapping
```

Return true if the mapping code at position `i_sf` is any of the `SFMAP_MULTI` mappings.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
    procedure :: is_multi_mapping => sf_channel_is_multi_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
    function sf_channel_is_multi_mapping (channel, i_sf) result (flag)
      class(sf_channel_t), intent(in) :: channel
      integer, intent(in) :: i_sf
      logical :: flag
      select case (channel%map_code(i_sf))
```

```
      case (SFMAP_NONE, SFMAP_SINGLE)
         flag = .false.
      case default
         flag = .true.
      end select
   end function sf_channel_is_multi_mapping
```

Return true if there is any nontrivial mapping in any of the channels.

Note: we provide an explicit public function. gfortran 4.6.3 has problems with the alternative implementation as a type-bound procedure for an array base object.

⟨*SF mappings: public*⟩+≡
```
   public :: any_sf_channel_has_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
   function any_sf_channel_has_mapping (channel) result (flag)
     type(sf_channel_t), dimension(:), intent(in) :: channel
     logical :: flag
     integer :: c
     flag = .false.
     do c = 1, size (channel)
        flag = flag .or. any (channel(c)%map_code /= SFMAP_NONE)
     end do
   end function any_sf_channel_has_mapping
```

Set a parameter index for an active multi mapping. We assume that the index array is allocated properly.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
   procedure :: set_par_index => sf_channel_set_par_index
```

⟨*SF mappings: procedures*⟩+≡
```
   subroutine sf_channel_set_par_index (channel, j, i_par)
     class(sf_channel_t), intent(inout) :: channel
     integer, intent(in) :: j
     integer, intent(in) :: i_par
     call channel%multi_mapping%set_index (j, i_par)
   end subroutine sf_channel_set_par_index
```

10.2.16   Unit tests

⟨*SF mappings: public*⟩+≡
```
   public :: sf_mappings_test
```

⟨*SF mappings: tests*⟩≡
```
   subroutine sf_mappings_test (u, results)
     integer, intent(in) :: u
     type(test_results_t), intent(inout) :: results
```
⟨*SF mappings: execute tests*⟩
```
   end subroutine sf_mappings_test
```

Check standard mapping

Probe the standard mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩≡
```
    call test (sf_mappings_1, "sf_mappings_1", &
        "standard pair mapping", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_1 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p

      write (u, "(A)")  "* Test output: sf_mappings_1"
      write (u, "(A)")  "*   Purpose: probe standard mapping"
      write (u, "(A)")

      allocate (sf_s_mapping_t :: mapping)
      select type (mapping)
      type is (sf_s_mapping_t)
         call mapping%init ()
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0):"
      p = [0._default, 0._default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.5):"
      p = [0.1_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.1):"
      p = [0.1_default, 0.1_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)
```

```
      allocate (sf_s_mapping_t :: mapping)
      select type (mapping)
      type is (sf_s_mapping_t)
         call mapping%init (power=2._default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      write (u, *)
      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0):"
      p = [0._default, 0._default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.5):"
      p = [0.1_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.1):"
      p = [0.1_default, 0.1_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)")  "I =", mapping%integral (100000)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_1"

    end subroutine sf_mappings_1
```

Channel entries

Construct channel entries and print them.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_2, "sf_mappings_2", &
        "structure-function mapping channels", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_2 (u)
      integer, intent(in) :: u
      type(sf_channel_t), dimension(:), allocatable :: channel
      integer :: c
```

```
      write (u, "(A)")  "* Test output: sf_mappings_2"
      write (u, "(A)")  "*   Purpose: construct and display &
           &mapping-channel objects"
      write (u, "(A)")

      call allocate_sf_channels (channel, n_channel = 6, n_strfun = 2)
      call channel(2)%activate_mapping ([1])
      call channel(3)%set_s_mapping ([1,2])
      call channel(4)%set_s_mapping ([1,2], power=2._default)
      call channel(5)%set_res_mapping ([1,2], m = 0.5_default, w = 0.1_default)
      call channel(6)%set_os_mapping ([1,2], m = 0.5_default)

      call channel(3)%set_par_index (1, 1)
      call channel(3)%set_par_index (2, 4)

      call channel(4)%set_par_index (1, 1)
      call channel(4)%set_par_index (2, 4)

      call channel(5)%set_par_index (1, 1)
      call channel(5)%set_par_index (2, 3)

      call channel(6)%set_par_index (1, 1)
      call channel(6)%set_par_index (2, 2)

      do c = 1, size (channel)
         write (u, "(I0,':')", advance="no")  c
         call channel(c)%write (u)
      end do

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_2"

   end subroutine sf_mappings_2
```

Check resonance mapping

Probe the resonance mapping of the unit square for different parameter values.
Also calculates integrals. For a finite number of bins, they differ slightly from
1, but the result is well-defined because we are not using random points.

The resonance mass is at $1/2$ the energy, the width is $1/10$.

⟨*SF mappings: execute tests*⟩+≡
```
      call test (sf_mappings_3, "sf_mappings_3", &
           "resonant pair mapping", &
           u, results)
```

⟨*SF mappings: tests*⟩+≡
```
      subroutine sf_mappings_3 (u)
        integer, intent(in) :: u
        class(sf_mapping_t), allocatable :: mapping
        real(default), dimension(2) :: p

        write (u, "(A)")  "* Test output: sf_mappings_3"
```

```
write (u, "(A)")  "*   Purpose: probe resonance pair mapping"
write (u, "(A)")

allocate (sf_res_mapping_t :: mapping)
select type (mapping)
type is (sf_res_mapping_t)
   call mapping%init (0.5_default, 0.1_default)
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, *)
write (u, "(A)")  "Probe at (0,0):"
p = [0._default, 0._default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.1,0.5):"
p = [0.1_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.1,0.1):"
p = [0.1_default, 0.1_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_mappings_3"

end subroutine sf_mappings_3
```

Check on-shell mapping

Probe the on-shell mapping of the unit square for different parameter values. Also calculates integrals. In this case, the Jacobian is constant and given by $|\log m^2|$, so this is also the value of the integral. The factor results from the variable change in the δ function $\delta(m^2 - x_1 x_2)$ which multiplies the cross section for the case at hand.

For the test, the (rescaled) resonance mass is set at $1/2$ the energy.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_4, "sf_mappings_4", &
        "on-shell pair mapping", &
        u, results)
```
⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_4 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p

      write (u, "(A)")  "* Test output: sf_mappings_4"
      write (u, "(A)")  "*   Purpose: probe on-shell pair mapping"
      write (u, "(A)")

      allocate (sf_os_mapping_t :: mapping)
      select type (mapping)
      type is (sf_os_mapping_t)
         call mapping%init (0.5_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0):"
      p = [0._default, 0._default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0,0.1):"
      p = [0._default, 0.1_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0,1.0):"
      p = [0._default, 1.0_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_4"

    end subroutine sf_mappings_4
```

Check endpoint mapping

Probe the endpoint mapping of the unit square for different parameter values.
Also calculates integrals. For a finite number of bins, they differ slightly from
1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_5, "sf_mappings_5", &
        "endpoint pair mapping", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_5 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p

      write (u, "(A)")  "* Test output: sf_mappings_5"
      write (u, "(A)")  "*   Purpose: probe endpoint pair mapping"
      write (u, "(A)")

      allocate (sf_ep_mapping_t :: mapping)
      select type (mapping)
      type is (sf_ep_mapping_t)
         call mapping%init ()
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0):"
      p = [0._default, 0._default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.5):"
      p = [0.1_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.7,0.2):"
      p = [0.7_default, 0.2_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Compute integral:"
```

1063

```
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_5"

    end subroutine sf_mappings_5
```

Check endpoint resonant mapping

Probe the endpoint mapping with resonance. Also calculates integrals.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_6, "sf_mappings_6", &
        "endpoint resonant mapping", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_6 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p

      write (u, "(A)")  "* Test output: sf_mappings_6"
      write (u, "(A)")  "*   Purpose: probe endpoint resonant mapping"
      write (u, "(A)")

      allocate (sf_epr_mapping_t :: mapping)
      select type (mapping)
      type is (sf_epr_mapping_t)
         call mapping%init (a = 1._default, m = 0.5_default, w = 0.1_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0):"
      p = [0._default, 0._default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.5):"
      p = [0.1_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
```

```
write (u, "(A)")  "Probe at (0.7,0.2):"
p = [0.7_default, 0.2_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)

write (u, "(A)")
write (u, "(A)")  "* Same mapping without resonance:"
write (u, "(A)")

allocate (sf_epr_mapping_t :: mapping)
select type (mapping)
type is (sf_epr_mapping_t)
   call mapping%init (a = 1._default)
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, *)
write (u, "(A)")  "Probe at (0,0):"
p = [0._default, 0._default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.1,0.5):"
p = [0.1_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.2):"
p = [0.7_default, 0.2_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_mappings_6"

end subroutine sf_mappings_6
```

Check endpoint on-shell mapping

Probe the endpoint mapping with an on-shell particle. Also calculates integrals.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_7, "sf_mappings_7", &
        "endpoint on-shell mapping", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_7 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p

      write (u, "(A)")  "* Test output: sf_mappings_7"
      write (u, "(A)")  "*   Purpose: probe endpoint on-shell mapping"
      write (u, "(A)")

      allocate (sf_epo_mapping_t :: mapping)
      select type (mapping)
      type is (sf_epo_mapping_t)
         call mapping%init (a = 1._default, m = 0.5_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0):"
      p = [0._default, 0._default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.1,0.5):"
      p = [0.1_default, 0.5_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Probe at (0.7,0.2):"
      p = [0.7_default, 0.2_default]
      call mapping%check (u, p, 1-p, "F7.5")

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)
```

```
      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_7"

    end subroutine sf_mappings_7
```

Check power mapping

Probe the power mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_8, "sf_mappings_8", &
         "power pair mapping", &
         u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_8 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p, pb

      write (u, "(A)")  "* Test output: sf_mappings_8"
      write (u, "(A)")  "*   Purpose: probe power pair mapping"
      write (u, "(A)")

      allocate (sf_ip_mapping_t :: mapping)
      select type (mapping)
      type is (sf_ip_mapping_t)
         call mapping%init (eps = 0.1_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0.5):"
      p = [0._default, 0.5_default]
      pb= [1._default, 0.5_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.5,0.5):"
      p = [0.5_default, 0.5_default]
      pb= [0.5_default, 0.5_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.9,0.5):"
      p = [0.9_default, 0.5_default]
      pb= [0.1_default, 0.5_default]
```

```
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.7,0.2):"
      p = [0.7_default, 0.2_default]
      pb= [0.3_default, 0.8_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.7,0.8):"
      p = [0.7_default, 0.8_default]
      pb= [0.3_default, 0.2_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.99,0.02):"
      p = [0.99_default, 0.02_default]
      pb= [0.01_default, 0.98_default]
      call mapping%check (u, p, pb, FMT_14, FMT_12)

      write (u, *)
      write (u, "(A)")  "Probe at (0.99,0.98):"
      p = [0.99_default, 0.98_default]
      pb= [0.01_default, 0.02_default]
      call mapping%check (u, p, pb, FMT_14, FMT_12)

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_8"

    end subroutine sf_mappings_8
```

Check resonant power mapping

Probe the power mapping of the unit square, adapted for an s-channel resonance,
for different parameter values. Also calculates integrals. For a finite number of
bins, they differ slightly from 1, but the result is well-defined because we are
not using random points.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_9, "sf_mappings_9", &
        "power resonance mapping", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_9 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p, pb
```

```
write (u, "(A)")  "* Test output: sf_mappings_9"
write (u, "(A)")  "*   Purpose: probe power resonant pair mapping"
write (u, "(A)")

allocate (sf_ipr_mapping_t :: mapping)
select type (mapping)
type is (sf_ipr_mapping_t)
   call mapping%init (eps = 0.1_default, m = 0.5_default, w = 0.1_default)
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, *)
write (u, "(A)")  "Probe at (0,0.5):"
p = [0._default, 0.5_default]
pb= [1._default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
pb= [0.5_default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.9,0.5):"
p = [0.9_default, 0.5_default]
pb= [0.1_default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.2):"
p = [0.7_default, 0.2_default]
pb= [0.3_default, 0.8_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.8):"
p = [0.7_default, 0.8_default]
pb= [0.3_default, 0.2_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.9999,0.02):"
p = [0.9999_default, 0.02_default]
pb= [0.0001_default, 0.98_default]
call mapping%check (u, p, pb, FMT_15, FMT_12)

write (u, *)
write (u, "(A)")  "Probe at (0.9999,0.98):"
p = [0.9999_default, 0.98_default]
```

1069

```
pb= [0.0001_default, 0.02_default]
call mapping%check (u, p, pb, FMT_15, FMT_12)

write (u, *)
write (u, "(A)")  "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)

write (u, "(A)")
write (u, "(A)")  "* Same mapping without resonance:"
write (u, "(A)")

allocate (sf_ipr_mapping_t :: mapping)
select type (mapping)
type is (sf_ipr_mapping_t)
   call mapping%init (eps = 0.1_default)
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, *)
write (u, "(A)")  "Probe at (0,0.5):"
p = [0._default, 0.5_default]
pb= [1._default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
pb= [0.5_default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.9,0.5):"
p = [0.9_default, 0.5_default]
pb= [0.1_default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.2):"
p = [0.7_default, 0.2_default]
pb= [0.3_default, 0.8_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.8):"
p = [0.7_default, 0.8_default]
pb= [0.3_default, 0.2_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
```

```
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_9"

    end subroutine sf_mappings_9
```

Check on-shell power mapping

Probe the power mapping of the unit square, adapted for single-particle production, for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡
```
      call test (sf_mappings_10, "sf_mappings_10", &
           "power on-shell mapping", &
           u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_10 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(2) :: p, pb

      write (u, "(A)")  "* Test output: sf_mappings_10"
      write (u, "(A)")  "*   Purpose: probe power on-shell mapping"
      write (u, "(A)")

      allocate (sf_ipo_mapping_t :: mapping)
      select type (mapping)
      type is (sf_ipo_mapping_t)
         call mapping%init (eps = 0.1_default, m = 0.5_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0.5):"
      p = [0._default, 0.5_default]
      pb= [1._default, 0.5_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0,0.02):"
      p = [0._default, 0.02_default]
      pb= [1._default, 0.98_default]
      call mapping%check (u, p, pb, FMT_15, FMT_12)
```

```
      write (u, *)
      write (u, "(A)")  "Probe at (0,0.98):"
      p = [0._default, 0.98_default]
      pb= [1._default, 0.02_default]
      call mapping%check (u, p, pb, FMT_15, FMT_12)

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_10"

    end subroutine sf_mappings_10
```

Check combined endpoint-power mapping

Probe the mapping for the beamstrahlung/ISR combination.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_11, "sf_mappings_11", &
        "endpoint/power combined mapping", &
        u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_11 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(4) :: p, pb

      write (u, "(A)")  "* Test output: sf_mappings_11"
      write (u, "(A)")  "*   Purpose: probe power pair mapping"
      write (u, "(A)")

      allocate (sf_ei_mapping_t :: mapping)
      select type (mapping)
      type is (sf_ei_mapping_t)
         call mapping%init (eps = 0.1_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
         call mapping%set_index (3, 3)
         call mapping%set_index (4, 4)
      end select

      call mapping%write (u)

      write (u, *)
      write (u, "(A)")  "Probe at (0.5, 0.5, 0.5, 0.5):"
      p = [0.5_default, 0.5_default, 0.5_default, 0.5_default]
      pb= [0.5_default, 0.5_default, 0.5_default, 0.5_default]
      call mapping%check (u, p, pb, FMT_16)
```

```
      write (u, *)
      write (u, "(A)")  "Probe at (0.7, 0.2, 0.4, 0.8):"
      p = [0.7_default, 0.2_default, 0.4_default, 0.8_default]
      pb= [0.3_default, 0.8_default, 0.6_default, 0.2_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.9, 0.06, 0.95, 0.1):"
      p = [0.9_default, 0.06_default, 0.95_default, 0.1_default]
      pb= [0.1_default, 0.94_default, 0.05_default, 0.9_default]
      call mapping%check (u, p, pb, FMT_13, FMT_12)

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_11"

    end subroutine sf_mappings_11
```

Check resonant endpoint-power mapping

Probe the mapping for the beamstrahlung/ISR combination.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_12, "sf_mappings_12", &
         "endpoint/power resonant combined mapping", &
         u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_12 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(4) :: p, pb

      write (u, "(A)")  "* Test output: sf_mappings_12"
      write (u, "(A)")  "*   Purpose: probe resonant combined mapping"
      write (u, "(A)")

      allocate (sf_eir_mapping_t :: mapping)
      select type (mapping)
      type is (sf_eir_mapping_t)
         call mapping%init (a = 1._default, &
              eps = 0.1_default, m = 0.5_default, w = 0.1_default)
         call mapping%set_index (1, 1)
         call mapping%set_index (2, 2)
         call mapping%set_index (3, 3)
         call mapping%set_index (4, 4)
      end select

      call mapping%write (u)
```

1073

```
      write (u, *)
      write (u, "(A)")  "Probe at (0.5, 0.5, 0.5, 0.5):"
      p = [0.5_default, 0.5_default, 0.5_default, 0.5_default]
      pb= [0.5_default, 0.5_default, 0.5_default, 0.5_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.7, 0.2, 0.4, 0.8):"
      p = [0.7_default, 0.2_default, 0.4_default, 0.8_default]
      pb= [0.3_default, 0.8_default, 0.6_default, 0.2_default]
      call mapping%check (u, p, pb, FMT_16)

      write (u, *)
      write (u, "(A)")  "Probe at (0.9, 0.06, 0.95, 0.1):"
      p = [0.9_default, 0.06_default, 0.95_default, 0.1_default]
      pb= [0.1_default, 0.94_default, 0.05_default, 0.9_default]
      call mapping%check (u, p, pb, FMT_15, FMT_12)

      write (u, *)
      write (u, "(A)")  "Compute integral:"
      write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

      deallocate (mapping)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_12"

    end subroutine sf_mappings_12
```

Check on-shell endpoint-power mapping

Probe the mapping for the beamstrahlung/ISR combination.

⟨*SF mappings: execute tests*⟩+≡
```
    call test (sf_mappings_13, "sf_mappings_13", &
         "endpoint/power on-shell combined mapping", &
         u, results)
```

⟨*SF mappings: tests*⟩+≡
```
    subroutine sf_mappings_13 (u)
      integer, intent(in) :: u
      class(sf_mapping_t), allocatable :: mapping
      real(default), dimension(4) :: p, pb

      write (u, "(A)")  "* Test output: sf_mappings_13"
      write (u, "(A)")  "*   Purpose: probe on-shell combined mapping"
      write (u, "(A)")

      allocate (sf_eio_mapping_t :: mapping)
      select type (mapping)
      type is (sf_eio_mapping_t)
         call mapping%init (a = 1._default, eps = 0.1_default, m = 0.5_default)
         call mapping%set_index (1, 1)
```

```
      call mapping%set_index (2, 2)
      call mapping%set_index (3, 3)
      call mapping%set_index (4, 4)
   end select

   call mapping%write (u)

   write (u, *)
   write (u, "(A)")  "Probe at (0.5, 0.5, 0.5, 0.5):"
   p = [0.5_default, 0.5_default, 0.5_default, 0.5_default]
   pb= [0.5_default, 0.5_default, 0.5_default, 0.5_default]
   call mapping%check (u, p, pb, FMT_16)

   write (u, *)
   write (u, "(A)")  "Probe at (0.7, 0.2, 0.4, 0.8):"
   p = [0.7_default, 0.2_default, 0.4_default, 0.8_default]
   pb= [0.3_default, 0.8_default, 0.6_default, 0.2_default]
   call mapping%check (u, p, pb, FMT_16)

   write (u, *)
   write (u, "(A)")  "Probe at (0.9, 0.06, 0.95, 0.1):"
   p = [0.9_default, 0.06_default, 0.95_default, 0.1_default]
   pb= [0.1_default, 0.94_default, 0.05_default, 0.9_default]
   call mapping%check (u, p, pb, FMT_14, FMT_12)

   write (u, *)
   write (u, "(A)")  "Compute integral:"
   write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

   deallocate (mapping)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: sf_mappings_13"

end subroutine sf_mappings_13
```

Check rescaling

Check the rescaling factor in on-shell basic mapping.

⟨*SF mappings: execute tests*⟩+≡
```
   call test (sf_mappings_14, "sf_mappings_14", &
       "rescaled on-shell mapping", &
       u, results)
```

⟨*SF mappings: tests*⟩+≡
```
   subroutine sf_mappings_14 (u)
     integer, intent(in) :: u
     real(default), dimension(2) :: p, r
     real(default) :: f, x_free, m2

     write (u, "(A)")  "* Test output: sf_mappings_14"
     write (u, "(A)")  "*   Purpose: probe rescaling in os mapping"
     write (u, "(A)")
```

1075

```
      p = [0.1_default, 0.2_default]
      x_free = 0.9_default
      m2 = 0.5_default

      call map_on_shell (r, f, p, -log (m2), x_free)

      write (u, "(A,9(1x," // FMT_14 // "))")  "p =", p
      write (u, "(A,9(1x," // FMT_14 // "))")  "r =", r
      write (u, "(A,9(1x," // FMT_14 // "))")  "f =", f
      write (u, "(A,9(1x," // FMT_14 // "))")  "*r=", x_free * product (r)

      write (u, *)

      call map_on_shell_inverse (r, f, p, -log (m2), x_free)

      write (u, "(A,9(1x," // FMT_14 // "))")  "p =", p
      write (u, "(A,9(1x," // FMT_14 // "))")  "r =", r
      write (u, "(A,9(1x," // FMT_14 // "))")  "f =", f
      write (u, "(A,9(1x," // FMT_14 // "))")  "*r=", x_free * product (r)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_mappings_14"

    end subroutine sf_mappings_14
```

10.3   Structure function base

⟨sf_base.f90⟩≡
  ⟨*File header*⟩

```
  module sf_base
```

⟨*Use kinds*⟩
⟨*Use strings*⟩
⟨*Use file utils*⟩
```
    use limits, only: FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use pdg_arrays
    use models
    use flavors
    use helicities
    use quantum_numbers
    use state_matrices
    use interactions
    use evaluators
    use particles
    use beams
    use sf_aux
```

```
    use sf_mappings

⟨Standard module head⟩

⟨SF base: public⟩

⟨SF base: parameters⟩

⟨SF base: types⟩

⟨SF base: interfaces⟩

⟨SF base: test types⟩

    contains

⟨SF base: procedures⟩

⟨SF base: tests⟩

    end module sf_base
```

10.3.1 Abstract structure-function data type

This type should hold all configuration data for a specific type of structure
function. The base object is empty; the implementations will fill it.

⟨*SF base: public*⟩≡
```
    public :: sf_data_t
```

⟨*SF base: types*⟩≡
```
    type, abstract :: sf_data_t
     contains
     ⟨SF base: sf data: TBP⟩
    end type sf_data_t
```

Output.

⟨*SF base: sf data: TBP*⟩≡
```
    procedure (sf_data_write), deferred :: write
```

⟨*SF base: interfaces*⟩≡
```
    abstract interface
      subroutine sf_data_write (data, unit, verbose)
        import
        class(sf_data_t), intent(in) :: data
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
      end subroutine sf_data_write
    end interface
```

Return true if this structure function is in generator mode. In that case, all
parameters are free, otherwise bound. (We do not support mixed cases.) Default
is: no generator.

⟨*SF base: sf data: TBP*⟩+≡

```
      procedure :: is_generator => sf_data_is_generator
```
⟨*SF base: procedures*⟩≡
```
    function sf_data_is_generator (data) result (flag)
      class(sf_data_t), intent(in) :: data
      logical :: flag
      flag = .false.
    end function sf_data_is_generator
```

Return the number of input parameters that determine the structure function.
⟨*SF base: sf data: TBP*⟩+≡
```
    procedure (sf_data_get_int), deferred :: get_n_par
```
⟨*SF base: interfaces*⟩+≡
```
    abstract interface
      function sf_data_get_int (data) result (n)
        import
        class(sf_data_t), intent(in) :: data
        integer :: n
      end function sf_data_get_int
    end interface
```

Return the outgoing particle PDG codes for the current setup. The codes can
be an array of particles, for each beam.
⟨*SF base: sf data: TBP*⟩+≡
```
    procedure (sf_data_get_pdg_out), deferred :: get_pdg_out
```
⟨*SF base: interfaces*⟩+≡
```
    abstract interface
      subroutine sf_data_get_pdg_out (data, pdg_out)
        import
        class(sf_data_t), intent(in) :: data
        type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      end subroutine sf_data_get_pdg_out
    end interface
```

Allocate a matching structure function interaction object and properly initialize
it.
⟨*SF base: sf data: TBP*⟩+≡
```
    procedure (sf_data_allocate_sf_int), deferred :: allocate_sf_int
```
⟨*SF base: interfaces*⟩+≡
```
    abstract interface
      subroutine sf_data_allocate_sf_int (data, sf_int)
        import
        class(sf_data_t), intent(in) :: data
        class(sf_int_t), intent(inout), allocatable :: sf_int
      end subroutine sf_data_allocate_sf_int
    end interface
```

Return the PDF set index, if applicable. We implement a default method which
returns zero. The PDF (builtin and LHA) implementations will override this.
⟨*SF base: sf data: TBP*⟩+≡
```
    procedure :: get_pdf_set => sf_data_get_pdf_set
```

⟨*SF base: procedures*⟩+≡
```
    function sf_data_get_pdf_set (data) result (pdf_set)
      class(sf_data_t), intent(in) :: data
      integer :: pdf_set
      pdf_set = 0
    end function sf_data_get_pdf_set
```

10.3.2 Structure-function chain configuration

This is the data type that the `processes` module uses for setting up its structure-function chain. For each structure function described by the beam data, there is an entry. The `i` array indicates the beam(s) to which this structure function applies, and the `data` object contains the actual configuration data.

⟨*SF base: public*⟩+≡
```
    public :: sf_config_t
```

⟨*SF base: types*⟩+≡
```
    type :: sf_config_t
      integer, dimension(:), allocatable :: i
      class(sf_data_t), allocatable :: data
     contains
     ⟨SF base: sf config: TBP⟩
    end type sf_config_t
```

Output:

⟨*SF base: sf config: TBP*⟩≡
```
    procedure :: write => sf_config_write
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_config_write (object, unit)
      class(sf_config_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (allocated (object%i)) then
        write (u, "(1x,A,2(1x,I0))")  "Structure-function configuration: &
             &beam(s)", object%i
        if (allocated (object%data))  call object%data%write (u)
      else
        write (u, "(1x,A)")  "Structure-function configuration: [undefined]"
      end if
    end subroutine sf_config_write
```

Initialize.

⟨*SF base: sf config: TBP*⟩+≡
```
    procedure :: init => sf_config_init
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_config_init (sf_config, i_beam, sf_data)
      class(sf_config_t), intent(out) :: sf_config
      integer, dimension(:), intent(in) :: i_beam
      class(sf_data_t), intent(in) :: sf_data
```

```
        allocate (sf_config%i (size (i_beam)), source = i_beam)
        allocate (sf_config%data, source = sf_data)
    end subroutine sf_config_init
```

Return the PDF set, if any.

⟨*SF base: sf config: TBP*⟩+≡
```
    procedure :: get_pdf_set => sf_config_get_pdf_set
```

⟨*SF base: procedures*⟩+≡
```
    function sf_config_get_pdf_set (sf_config) result (pdf_set)
      class(sf_config_t), intent(in) :: sf_config
      integer :: pdf_set
      pdf_set = sf_config%data%get_pdf_set ()
    end function sf_config_get_pdf_set
```

10.3.3 Structure-function instance

The `sf_int_t` data type contains an `interaction_t` object (it is an extension of this type) and a pointer to the `sf_data_t` configuration data. This interaction, or copies of it, is used to implement structure-function kinematics and dynamics in the context of process evaluation.

The status code `status` tells whether the interaction is undefined, has defined kinematics (but matrix elements invalid), or is completely defined. There is also a status code for failure. The implementation is responsible for updating the status.

The entries `mi2`, `mr2`, and `mo2` hold the squared invariant masses of the incoming, radiated, and outgoing particle, respectively. They are supposed to be set upon initialization, but could also be varied event by event.

If the radiated or outgoing mass is nonzero, we may need to apply an on-shell projection. The projection mode is stored as `on_shell_mode`.

The array `beam_index` is the list of beams on which this structure function applies (1, 2, or both). The arrays `incoming`, `radiated`, and `outgoing` contain the indices of the respective particle sets within the interaction, for convenient lookup. The array `par_index` indicates the MC input parameters that this entry will use up in the structure-function chain. The first parameter (or the first two, for a spectrum) in this array determines the momentum fraction and is thus subject to global mappings.

In the abstract base type, we do not implement the data pointer. This allows us to restrict its type in the implementations.

⟨*SF base: public*⟩+≡
```
    public :: sf_int_t
```

⟨*SF base: types*⟩+≡
```
    type, abstract, extends (interaction_t) :: sf_int_t
      integer :: status = SF_UNDEFINED
      real(default), dimension(:), allocatable :: mi2
      real(default), dimension(:), allocatable :: mr2
      real(default), dimension(:), allocatable :: mo2
      integer :: on_shell_mode = KEEP_ENERGY
      logical :: qmin_defined = .false.
      logical :: qmax_defined = .false.
```

```
    real(default), dimension(:), allocatable :: qmin
    real(default), dimension(:), allocatable :: qmax
    integer, dimension(:), allocatable :: beam_index
    integer, dimension(:), allocatable :: incoming
    integer, dimension(:), allocatable :: radiated
    integer, dimension(:), allocatable :: outgoing
    integer, dimension(:), allocatable :: par_index
    integer, dimension(:), allocatable :: par_primary
  contains
  ⟨SF base: sf int: TBP⟩
  end type sf_int_t
```

Status codes. The codes that refer to links, masks, and connections, apply to structure-function chains only.

The status codes are public.

⟨*SF base: parameters*⟩≡
```
    integer, parameter, public :: SF_UNDEFINED = 0
    integer, parameter, public :: SF_INITIAL = 1
    integer, parameter, public :: SF_DONE_LINKS = 2
    integer, parameter, public :: SF_FAILED_MASK = 3
    integer, parameter, public :: SF_DONE_MASK = 4
    integer, parameter, public :: SF_FAILED_CONNECTIONS = 5
    integer, parameter, public :: SF_DONE_CONNECTIONS = 6
    integer, parameter, public :: SF_SEED_KINEMATICS = 10
    integer, parameter, public :: SF_FAILED_KINEMATICS = 11
    integer, parameter, public :: SF_DONE_KINEMATICS = 12
    integer, parameter, public :: SF_FAILED_EVALUATION = 13
    integer, parameter, public :: SF_EVALUATED = 20
```

Write a string version of the status code:

⟨*SF base: procedures*⟩+≡
```
    subroutine write_sf_status (status, u)
      integer, intent(in) :: status
      integer, intent(in) :: u
      select case (status)
      case (SF_UNDEFINED)
         write (u, "(1x,'[',A,']')")  "undefined"
      case (SF_INITIAL)
         write (u, "(1x,'[',A,']')")  "initialized"
      case (SF_DONE_LINKS)
         write (u, "(1x,'[',A,']')")  "links set"
      case (SF_FAILED_MASK)
         write (u, "(1x,'[',A,']')")  "mask mismatch"
      case (SF_DONE_MASK)
         write (u, "(1x,'[',A,']')")  "mask set"
      case (SF_FAILED_CONNECTIONS)
         write (u, "(1x,'[',A,']')")  "connections failed"
      case (SF_DONE_CONNECTIONS)
         write (u, "(1x,'[',A,']')")  "connections set"
      case (SF_SEED_KINEMATICS)
         write (u, "(1x,'[',A,']')")  "incoming momenta set"
      case (SF_FAILED_KINEMATICS)
         write (u, "(1x,'[',A,']')")  "kinematics failed"
```

```
      case (SF_DONE_KINEMATICS)
         write (u, "(1x,'[',A,']')")  "kinematics set"
      case (SF_FAILED_EVALUATION)
         write (u, "(1x,'[',A,']')")  "evaluation failed"
      case (SF_EVALUATED)
         write (u, "(1x,'[',A,']')")  "evaluated"
      end select
   end subroutine write_sf_status
```

Finalizer. Required because the base interaction needs a finalizer. We provide
an implementation, which need not be overwritten if the implementation does
not add finalizable subobjects.

⟨*SF base: sf int: TBP*⟩≡
```
   procedure :: final => sf_int_final
```

⟨*SF base: procedures*⟩+≡
```
   subroutine sf_int_final (object)
     class(sf_int_t), intent(inout) :: object
     call interaction_final (object%interaction_t)
   end subroutine sf_int_final
```

This is the basic output routine. Display status and interaction.

⟨*SF base: sf int: TBP*⟩+≡
```
   procedure :: base_write => sf_int_base_write
```

⟨*SF base: procedures*⟩+≡
```
   subroutine sf_int_base_write (object, unit, testflag)
     class(sf_int_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: testflag
     integer :: u
     u = output_unit (unit)
     write (u, "(1x,A)", advance="no")  "SF instance:"
     call write_sf_status (object%status, u)
     if (allocated (object%beam_index)) &
         write (u, "(3x,A,2(1x,I0))")  "beam      =", object%beam_index
     if (allocated (object%incoming)) &
         write (u, "(3x,A,2(1x,I0))")  "incoming  =", object%incoming
     if (allocated (object%radiated)) &
         write (u, "(3x,A,2(1x,I0))")  "radiated  =", object%radiated
     if (allocated (object%outgoing)) &
         write (u, "(3x,A,2(1x,I0))")  "outgoing  =", object%outgoing
     if (allocated (object%par_index)) &
         write (u, "(3x,A,2(1x,I0))")  "parameter =", object%par_index
     if (object%qmin_defined) &
         write (u, "(3x,A,1x," // FMT_19 // ")")  "q_min     =", object%qmin
     if (object%qmax_defined) &
         write (u, "(3x,A,1x," // FMT_19 // ")")  "q_max     =", object%qmax
     call interaction_write (object%interaction_t, u, testflag = testflag)
   end subroutine sf_int_base_write
```

The type string identifies the structure function class, and possibly more details
about the structure function.

⟨*SF base: sf int: TBP*⟩+≡

```
        procedure (sf_int_type_string), deferred :: type_string
```
⟨*SF base: interfaces*⟩+≡
```
    abstract interface
       function sf_int_type_string (object) result (string)
         import
         class(sf_int_t), intent(in) :: object
         type(string_t) :: string
       end function sf_int_type_string
    end interface
```

Output of the concrete object. We should not forget to call the output routine
for the base type.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure (sf_int_write), deferred :: write
```
⟨*SF base: interfaces*⟩+≡
```
    abstract interface
       subroutine sf_int_write (object, unit, testflag)
         import
         class(sf_int_t), intent(in) :: object
         integer, intent(in), optional :: unit
         logical, intent(in), optional :: testflag
       end subroutine sf_int_write
    end interface
```

Basic initialization: set the invariant masses for the particles and initialize the
interaction. The caller should then add states to the interaction and freeze it.

The dimension of the mask should be equal to the sum of the dimensions
of the mass-squared arrays, which determine incoming, radiated, and outgoing
particles, respectively.

Optionally, we can define minimum and maximum values for the momen-
tum transfer to the outgoing particle(s). If all masses are zero, this is actually
required for non-collinear splitting.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: base_init => sf_int_base_init
```
⟨*SF base: procedures*⟩+≡
```
    subroutine sf_int_base_init &
         (sf_int, mask, mi2, mr2, mo2, qmin, qmax, hel_lock)
      class(sf_int_t), intent(out) :: sf_int
      type (quantum_numbers_mask_t), dimension(:), intent(in) :: mask
      real(default), dimension(:), intent(in) :: mi2, mr2, mo2
      real(default), dimension(:), intent(in), optional :: qmin, qmax
      integer, dimension(:), intent(in), optional :: hel_lock
      allocate (sf_int%mi2 (size (mi2)))
      sf_int%mi2 = mi2
      allocate (sf_int%mr2 (size (mr2)))
      sf_int%mr2 = mr2
      allocate (sf_int%mo2 (size (mo2)))
      sf_int%mo2 = mo2
      if (present (qmin)) then
         sf_int%qmin_defined = .true.
         allocate (sf_int%qmin (size (qmin)))
```

```
              sf_int%qmin = qmin
          end if
          if (present (qmax)) then
              sf_int%qmax_defined = .true.
              allocate (sf_int%qmax (size (qmax)))
              sf_int%qmax = qmax
          end if
          call interaction_init (sf_int%interaction_t, &
                size (mi2), 0, size (mr2) + size (mo2), &
                mask = mask, hel_lock = hel_lock, set_relations = .true.)
        end subroutine sf_int_base_init
```

Set the indices of the incoming, radiated, and outgoing particles, respectively.

⟨*SF base: sf int: TBP*⟩+≡
```
      procedure :: set_incoming => sf_int_set_incoming
      procedure :: set_radiated => sf_int_set_radiated
      procedure :: set_outgoing => sf_int_set_outgoing
```

⟨*SF base: procedures*⟩+≡
```
      subroutine sf_int_set_incoming (sf_int, incoming)
        class(sf_int_t), intent(inout) :: sf_int
        integer, dimension(:), intent(in) :: incoming
        allocate (sf_int%incoming (size (incoming)))
        sf_int%incoming = incoming
      end subroutine sf_int_set_incoming

      subroutine sf_int_set_radiated (sf_int, radiated)
        class(sf_int_t), intent(inout) :: sf_int
        integer, dimension(:), intent(in) :: radiated
        allocate (sf_int%radiated (size (radiated)))
        sf_int%radiated = radiated
      end subroutine sf_int_set_radiated

      subroutine sf_int_set_outgoing (sf_int, outgoing)
        class(sf_int_t), intent(inout) :: sf_int
        integer, dimension(:), intent(in) :: outgoing
        allocate (sf_int%outgoing (size (outgoing)))
        sf_int%outgoing = outgoing
      end subroutine sf_int_set_outgoing
```

Initialization. This proceeds via an abstract data object, which for the actual implementation should have the matching concrete type. Since all implementations have the same signature, we can prepare a deferred procedure. The data object will become the target of a corresponding pointer within the `sf_int_t` implementation.

This should call the previous procedure.

⟨*SF base: sf int: TBP*⟩+≡
```
      procedure (sf_int_init), deferred :: init
```

⟨*SF base: interfaces*⟩+≡
```
      abstract interface
        subroutine sf_int_init (sf_int, data)
          import
```

```
        class(sf_int_t), intent(out) :: sf_int
        class(sf_data_t), intent(in), target :: data
      end subroutine sf_int_init
    end interface
```

Complete initialization. This routine contains initializations that can only be performed after the interaction object got its final shape, i.e., redundant helicities have been eliminated by matching with beams and process.

The default implementation does nothing.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: setup_constants => sf_int_setup_constants
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_int_setup_constants (sf_int)
      class(sf_int_t), intent(inout) :: sf_int
    end subroutine sf_int_setup_constants
```

Set beam indices, i.e., the beam(s) on which this structure function applies.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: set_beam_index => sf_int_set_beam_index
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_int_set_beam_index (sf_int, beam_index)
      class(sf_int_t), intent(inout) :: sf_int
      integer, dimension(:), intent(in) :: beam_index
      allocate (sf_int%beam_index (size (beam_index)))
      sf_int%beam_index = beam_index
    end subroutine sf_int_set_beam_index
```

Set parameter indices, indicating which MC input parameters are to be used for evaluating this structure function.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: set_par_index => sf_int_set_par_index
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_int_set_par_index (sf_int, par_index)
      class(sf_int_t), intent(inout) :: sf_int
      integer, dimension(:), intent(in) :: par_index
      allocate (sf_int%par_index (size (par_index)))
      sf_int%par_index = par_index
    end subroutine sf_int_set_par_index
```

Initialize the structure-function kinematics, setting incoming momenta. We assume that array shapes match.

Three versions. The first version relies on the momenta being linked to another interaction. The second version sets the momenta explicitly. In the third version, we first compute momenta for the specified energies and store those.

⟨*SF base: sf int: TBP*⟩+≡
```
    generic :: seed_kinematics => sf_int_receive_momenta
    generic :: seed_kinematics => sf_int_seed_momenta
    generic :: seed_kinematics => sf_int_seed_energies
```

```
      procedure :: sf_int_receive_momenta
      procedure :: sf_int_seed_momenta
      procedure :: sf_int_seed_energies
```
⟨*SF base: procedures*⟩+≡
```
    subroutine sf_int_receive_momenta (sf_int)
      class(sf_int_t), intent(inout) :: sf_int
      if (sf_int%status >= SF_INITIAL) then
         call interaction_receive_momenta (sf_int%interaction_t)
         sf_int%status = SF_SEED_KINEMATICS
      end if
    end subroutine sf_int_receive_momenta

    subroutine sf_int_seed_momenta (sf_int, k)
      class(sf_int_t), intent(inout) :: sf_int
      type(vector4_t), dimension(:), intent(in) :: k
      if (sf_int%status >= SF_INITIAL) then
         call interaction_set_momenta (sf_int%interaction_t, k, &
              outgoing=.false.)
         sf_int%status = SF_SEED_KINEMATICS
      end if
    end subroutine sf_int_seed_momenta

    subroutine sf_int_seed_energies (sf_int, E)
      class(sf_int_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: E
      type(vector4_t), dimension(:), allocatable :: k
      integer :: j
      if (sf_int%status >= SF_INITIAL) then
         allocate (k (size (E)))
         if (all (E**2 >= sf_int%mi2)) then
            do j = 1, size (E)
               k(j) = vector4_moving (E(j), &
                    (3-2*j) * sqrt (E(j)**2 - sf_int%mi2(j)), 3)
            end do
            call sf_int%seed_kinematics (k)
         end if
      end if
    end subroutine sf_int_seed_energies
```

Tell if in generator mode. By default, this is false. To be overridden where
appropriate; we may refer to the **is_generator** method of the **data** component
in the concrete type.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: is_generator => sf_int_is_generator
```
⟨*SF base: procedures*⟩+≡
```
    function sf_int_is_generator (sf_int) result (flag)
      class(sf_int_t), intent(in) :: sf_int
      logical :: flag
      flag = .false.
    end function sf_int_is_generator
```

Generate free parameters `r`. Parameters are free if they do not correspond to integration parameters (i.e., are bound), but are generated by the structure function object itself. By default, all parameters are bound, and the output values of this procedure will be discarded. With free parameters, we have to override this procedure.

The value `x_free` is the renormalization factor of the total energy that corresponds to the free parameters. If there are no free parameters, the procedure will not change its value, which starts as unity. Otherwise, the fraction is typically decreased, but may also be increased in some cases.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure :: generate_free => sf_int_generate_free
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_int_generate_free (sf_int, r, rb,  x_free)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(inout) :: x_free
  r = 0
  rb= 1
end subroutine sf_int_generate_free
```

Complete the structure-function kinematics, derived from an input parameter (array) r between 0 and 1. The interaction momenta are calculated, and we return x (the momentum fraction), and f (the Jacobian factor for the map $r \to x$), if `map` is set.

If the `map` flag is unset, r and x values will coincide, and f will become unity. If it is set, the structure-function implementation chooses a convenient mapping from r to x with Jacobian f.

In the `inverse_kinematics` variant, we exchange the intent of `x` and `r`. The momenta are calculated only if the optional flag `set_momenta` is present and set.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure (sf_int_complete_kinematics), deferred :: complete_kinematics
procedure (sf_int_inverse_kinematics), deferred :: inverse_kinematics
```

⟨*SF base: interfaces*⟩+≡
```
abstract interface
   subroutine sf_int_complete_kinematics (sf_int, x, f, r, rb, map)
     import
     class(sf_int_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(out) :: x
     real(default), intent(out) :: f
     real(default), dimension(:), intent(in) :: r
     real(default), dimension(:), intent(in) :: rb
     logical, intent(in) :: map
   end subroutine sf_int_complete_kinematics
end interface

abstract interface
   subroutine sf_int_inverse_kinematics (sf_int, x, f, r, rb, map, &
        set_momenta)
     import
     class(sf_int_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(in) :: x
```

```
          real(default), intent(out) :: f
          real(default), dimension(:), intent(out) :: r
          real(default), dimension(:), intent(out) :: rb
          logical, intent(in) :: map
          logical, intent(in), optional :: set_momenta
        end subroutine sf_int_inverse_kinematics
      end interface
```

Single splitting: compute momenta, given x input parameters. We assume that the incoming momentum is set. The status code is set to `SF_FAILED_KINEMATICS` if the x array does not correspond to a valid momentum configuration. Otherwise, it is updated to `SF_DONE_KINEMATICS`.

We force the outgoing particle on-shell. The on-shell projection is determined by the `on_shell_mode`. The radiated particle should already be on shell.

⟨*SF base: sf int: TBP*⟩+≡
```
      procedure :: split_momentum => sf_int_split_momentum
```

⟨*SF base: procedures*⟩+≡
```
      subroutine sf_int_split_momentum (sf_int, x, xb1)
        class(sf_int_t), intent(inout) :: sf_int
        real(default), dimension(:), intent(in) :: x
        real(default), intent(in) :: xb1
        type(vector4_t) :: k
        type(vector4_t), dimension(2) :: q
        type(splitting_data_t) :: sd
        real(default) :: E1, E2
        logical :: fail
        if (sf_int%status >= SF_SEED_KINEMATICS) then
           k = interaction_get_momentum (sf_int%interaction_t, 1)
           call sd%init (k, &
                sf_int%mi2(1), sf_int%mr2(1), sf_int%mo2(1), &
                collinear = size (x) == 1)
           call sd%set_t_bounds (x(1), xb1)
           select case (size (x))
           case (1)
           case (3)
              if (sf_int%qmax_defined) then
                 if (sf_int%qmin_defined) then
                    call sd%sample_t (x(2), &
                         t0 = - sf_int%qmax(1) ** 2, t1 = - sf_int%qmin(1) ** 2)
                 else
                    call sd%sample_t (x(2), &
                         t0 = - sf_int%qmax(1) ** 2)
                 end if
              else
                 if (sf_int%qmin_defined) then
                    call sd%sample_t (x(2), t1 = - sf_int%qmin(1) ** 2)
                 else
                    call sd%sample_t (x(2))
                 end if
              end if
              call sd%sample_phi (x(3))
           case default
              call msg_bug ("Structure function: impossible number of parameters")
```

```
      end select
      q = sd%split_momentum (k)
      call on_shell (q, [sf_int%mr2, sf_int%mo2], &
            sf_int%on_shell_mode)
      call interaction_set_momenta (sf_int%interaction_t, &
            q, outgoing=.true.)
      E1 = energy (q(1))
      E2 = energy (q(2))
      fail = E1 < 0 .or. E2 < 0 &
            .or. E1 ** 2 < sf_int%mr2(1) &
            .or. E2 ** 2 < sf_int%mo2(1)
      if (fail) then
         sf_int%status = SF_FAILED_KINEMATICS
      else
         sf_int%status = SF_DONE_KINEMATICS
      end if
   end if
 end subroutine sf_int_split_momentum
```

Pair splitting: two incoming momenta, two radiated, two outgoing. This is simple because we insist on all momenta being collinear.

⟨*SF base: sf int: TBP*⟩+≡

```
   procedure :: split_momenta => sf_int_split_momenta
```

⟨*SF base: procedures*⟩+≡

```
   subroutine sf_int_split_momenta (sf_int, x, xb1)
     class(sf_int_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(in) :: x
     real(default), dimension(:), intent(in) :: xb1
     type(vector4_t), dimension(2) :: k
     type(vector4_t), dimension(4) :: q
     real(default), dimension(4) :: E
     logical :: fail
     if (sf_int%status >= SF_SEED_KINEMATICS) then
        select case (size (x))
        case (2)
        case default
           call msg_bug ("Pair structure function: recoil requested &
                 &but not implemented yet")
        end select
        k(1) = interaction_get_momentum (sf_int%interaction_t, 1)
        k(2) = interaction_get_momentum (sf_int%interaction_t, 2)
        q(1:2) = xb1 * k
        q(3:4) = x * k
        select case (size (sf_int%mr2))
        case (2)
           call on_shell (q, &
                 [sf_int%mr2(1), sf_int%mr2(2), &
                 sf_int%mo2(1), sf_int%mo2(2)], &
                 sf_int%on_shell_mode)
           call interaction_set_momenta (sf_int%interaction_t, &
                 q, outgoing=.true.)
           E = energy (q)
           fail = any (E < 0) &
```

```
                    .or. any (E(1:2) ** 2 < sf_int%mr2) &
                    .or. any (E(3:4) ** 2 < sf_int%mo2)
          case default;  call msg_bug ("split momenta: incorrect use")
          end select
          if (fail) then
             sf_int%status = SF_FAILED_KINEMATICS
          else
             sf_int%status = SF_DONE_KINEMATICS
          end if
       end if
    end subroutine sf_int_split_momenta
```

Pair spectrum: the reduced version of the previous splitting, without radiated
momenta.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: reduce_momenta => sf_int_reduce_momenta
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_int_reduce_momenta (sf_int, x)
      class(sf_int_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      type(vector4_t), dimension(2) :: k
      type(vector4_t), dimension(2) :: q
      real(default), dimension(2) :: E
      logical :: fail
      if (sf_int%status >= SF_SEED_KINEMATICS) then
         select case (size (x))
         case (2)
         case default
            call msg_bug ("Pair spectrum: recoil requested &
                 &but not implemented yet")
         end select
         k(1) = interaction_get_momentum (sf_int%interaction_t, 1)
         k(2) = interaction_get_momentum (sf_int%interaction_t, 2)
         q = x * k
         call on_shell (q, &
              [sf_int%mo2(1), sf_int%mo2(2)], &
              sf_int%on_shell_mode)
         call interaction_set_momenta (sf_int%interaction_t, &
              q, outgoing=.true.)
         E = energy (q)
         fail = any (E < 0) &
              .or. any (E ** 2 < sf_int%mo2)
         if (fail) then
            sf_int%status = SF_FAILED_KINEMATICS
         else
            sf_int%status = SF_DONE_KINEMATICS
         end if
      end if
    end subroutine sf_int_reduce_momenta
```

The inverse procedure: we compute the x array from the momentum configura-
tion. In an overriding TBP, we may also set internal data that depend on this,
for convenience.

NOTE: Here and above, the single-particle case is treated in detail, allowing for non-collinearity and non-vanishing masses and nontrivial momentum-transfer bounds. For the pair case, we currently implement only collinear splitting and assume massless particles. This should be improved.

NOTE: We should also improve this in the sense of extracting both x and $\bar{x}$, where for the latter we may consult the beam remnant. The current implementation is not sensitive to small $\bar{x}$ values.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure :: recover_x => sf_int_recover_x
procedure :: base_recover_x => sf_int_recover_x
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_int_recover_x (sf_int, x, x_free)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(inout), optional :: x_free
  type(vector4_t), dimension(:), allocatable :: k
  type(vector4_t), dimension(:), allocatable :: q
  type(splitting_data_t) :: sd
  if (sf_int%status >= SF_SEED_KINEMATICS) then
      allocate (k (interaction_get_n_in (sf_int%interaction_t)))
      allocate (q (interaction_get_n_out (sf_int%interaction_t)))
      k = interaction_get_momenta (sf_int%interaction_t, outgoing=.false.)
      q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
      select case (size (k))
      case (1)
         call sd%init (k(1), &
              sf_int%mi2(1), sf_int%mr2(1), sf_int%mo2(1), &
              collinear = size (x) == 1)
         call sd%recover (k(1), q(2), sf_int%on_shell_mode)
         x(1) = sd%get_x ()
         select case (size (x))
         case (1)
         case (3)
            if (sf_int%qmax_defined) then
               if (sf_int%qmin_defined) then
                  call sd%inverse_t (x(2), &
                       t0 = - sf_int%qmax(1) ** 2, t1 = - sf_int%qmin(1) ** 2)
               else
                  call sd%inverse_t (x(2), &
                       t0 = - sf_int%qmax(1) ** 2)
               end if
            else
               if (sf_int%qmin_defined) then
                  call sd%inverse_t (x(2), t1 = - sf_int%qmin(1) ** 2)
               else
                  call sd%inverse_t (x(2))
               end if
            end if
            call sd%inverse_phi (x(3))
         case default
            call msg_bug ("Structure function: impossible number &
                 &of parameters")
         end select
```

1091

```
          case (2)
             select case (size (x))
             case (2)
             case default
                call msg_bug ("Pair structure function: recoil requested &
                     &but not implemented yet")
             end select
             select case (sf_int%on_shell_mode)
             case (KEEP_ENERGY)
                select case (size (q))
                case (4)
                   x = energy (q(3:4)) / energy (k)
                case (2)
                   x = energy (q) / energy (k)
                end select
             case (KEEP_MOMENTUM)
                select case (size (q))
                case (4)
                   x = longitudinal_part (q(3:4)) / longitudinal_part (k)
                case (2)
                   x = longitudinal_part (q) / longitudinal_part (k)
                end select
             end select
          end select
       end if
    end subroutine sf_int_recover_x
```

Apply the structure function, i.e., evaluate the interaction. For the calculation, we may use the stored momenta, any further information stored inside the `sf_int` implementation during kinematics setup, and the given energy scale. It may happen that for the given kinematics the value is not defined. This should be indicated by the status code.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure (sf_int_apply), deferred :: apply
```

⟨*SF base: interfaces*⟩+≡
```
    abstract interface
       subroutine sf_int_apply (sf_int, scale)
         import
         class(sf_int_t), intent(inout) :: sf_int
         real(default), intent(in) :: scale
       end subroutine sf_int_apply
    end interface
```

10.3.4   Accessing the structure function

Return metadata. Once `interaction_t` is rewritten in OO, some of this will be inherited.

The number of outgoing is equal to the number of incoming particles. The radiated particles are the difference.

⟨*SF base: sf int: TBP*⟩+≡

```
    procedure :: get_n_in => sf_int_get_n_in
    procedure :: get_n_rad => sf_int_get_n_rad
    procedure :: get_n_out => sf_int_get_n_out
```

⟨*SF base: procedures*⟩+≡
```
    function sf_int_get_n_in (sf_int) result (n_in)
      class(sf_int_t), intent(in) :: sf_int
      integer :: n_in
      n_in = interaction_get_n_in (sf_int%interaction_t)
    end function sf_int_get_n_in

    function sf_int_get_n_rad (sf_int) result (n_rad)
      class(sf_int_t), intent(in) :: sf_int
      integer :: n_rad
      n_rad = interaction_get_n_out (sf_int%interaction_t) &
            - interaction_get_n_in (sf_int%interaction_t)
    end function sf_int_get_n_rad

    function sf_int_get_n_out (sf_int) result (n_out)
      class(sf_int_t), intent(in) :: sf_int
      integer :: n_out
      n_out = interaction_get_n_in (sf_int%interaction_t)
    end function sf_int_get_n_out
```

Number of matrix element entries in the interaction:

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: get_n_states => sf_int_get_n_states
```

⟨*SF base: procedures*⟩+≡
```
    function sf_int_get_n_states (sf_int) result (n_states)
      class(sf_int_t), intent(in) :: sf_int
      integer :: n_states
      n_states = interaction_get_n_matrix_elements (sf_int%interaction_t)
    end function sf_int_get_n_states
```

Return a specific state as a quantum-number array.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: get_state => sf_int_get_state
```

⟨*SF base: procedures*⟩+≡
```
    function sf_int_get_state (sf_int, i) result (qn)
      class(sf_int_t), intent(in) :: sf_int
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      integer, intent(in) :: i
      allocate (qn (interaction_get_n_tot (sf_int%interaction_t)))
      qn = interaction_get_quantum_numbers (sf_int%interaction_t, i)
    end function sf_int_get_state
```

Return the matrix-element values for all states. We can assume that the matrix
elements are real, so we take the real part.

⟨*SF base: sf int: TBP*⟩+≡
```
    procedure :: get_values => sf_int_get_values
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_get_values (sf_int, value)
  class(sf_int_t), intent(in) :: sf_int
  real(default), dimension(:), intent(out) :: value
  integer :: i
  if (sf_int%status >= SF_EVALUATED) then
     do i = 1, size (value)
        value(i) = interaction_get_matrix_element &
             (sf_int%interaction_t, i)
     end do
  else
     value = 0
  end if
end subroutine sf_int_get_values
```

10.3.5 Direct calculations

Compute a structure function value (array) directly, given an array of x values and a scale. If the energy is also given, we initialize the kinematics for that energy, otherwise take it from a previous run.

We assume that the E array has dimension n_in, and the x array has n_par.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: compute_values => sf_int_compute_values
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_compute_values (sf_int, value, x, xb, scale, E)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: value
  real(default), dimension(:), intent(in) :: x
  real(default), dimension(:), intent(in) :: xb
  real(default), intent(in) :: scale
  real(default), dimension(:), intent(in), optional :: E
  real(default), dimension(size (x)) :: xx
  real(default) :: f
  if (present (E))  call sf_int%seed_kinematics (E)
  if (sf_int%status >= SF_SEED_KINEMATICS) then
     call sf_int%complete_kinematics (xx, f, x, xb, map=.false.)
     call sf_int%apply (scale)
     call sf_int%get_values (value)
     value = value * f
  else
     value = 0
  end if
end subroutine sf_int_compute_values
```

Compute just a single value for one of the states, i.e., throw the others away.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: compute_value => sf_int_compute_value
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_compute_value &
     (sf_int, i_state, value, x, xb, scale, E)
```

```
   class(sf_int_t), intent(inout) :: sf_int
   integer, intent(in) :: i_state
   real(default), intent(out) :: value
   real(default), dimension(:), intent(in) :: x
   real(default), dimension(:), intent(in) :: xb
   real(default), intent(in) :: scale
   real(default), dimension(:), intent(in), optional :: E
   real(default), dimension(:), allocatable :: value_array
   if (sf_int%status >= SF_INITIAL) then
      allocate (value_array (sf_int%get_n_states ()))
      call sf_int%compute_values (value_array, x, xb, scale, E)
      value = value_array(i_state)
   else
      value = 0
   end if
 end subroutine sf_int_compute_value
```

10.3.6 Structure-function instance

This is a wrapper for `sf_int_t` objects, such that we can build an array with different structure-function types. The structure-function contains an array (a sequence) of `sf_int_t` objects.

The object, it holds the evaluator that connects the preceding part of the structure-function chain to the current interaction.

It also stores the input and output parameter values for the contained structure function. The `r` array has a second dimension, corresponding to the mapping channels in a multi-channel configuration. There is a Jacobian entry `f` for each channel. The corresponding logical array `mapping` tells whether we apply the mapping appropriate for the current structure function in this channel. The `x` parameter values (energy fractions) are common to all channels.

⟨*SF base: types*⟩+≡
```
   type :: sf_instance_t
      class(sf_int_t), allocatable :: int
      type(evaluator_t) :: eval
      real(default), dimension(:,:), allocatable :: r
      real(default), dimension(:,:), allocatable :: rb
      real(default), dimension(:), allocatable :: f
      logical, dimension(:), allocatable :: m
      real(default), dimension(:), allocatable :: x
   end type sf_instance_t
```

10.3.7 Structure-function chain

A chain is an array of structure functions `sf`, initiated by a beam setup. We do not use this directly for evaluation, but create instances with copies of the contained interactions.

`n_par` is the total number of parameters that is necessary for completely determining the structure-function chain. `n_bound` is the number of MC input parameters that are requested from the integrator. The difference of `n_par` and

1095

`n_bound` is the number of free parameters, which are generated by a structure-function object in generator mode.

⟨*SF base: public*⟩+≡
```
    public :: sf_chain_t
```

⟨*SF base: types*⟩+≡
```
    type, extends (beam_t) :: sf_chain_t
      type(beam_data_t), pointer :: beam_data => null ()
      integer :: n_in = 0
      integer :: n_strfun = 0
      integer :: n_par = 0
      integer :: n_bound = 0
      type(sf_instance_t), dimension(:), allocatable :: sf
      logical :: trace_enable = .false.
      integer :: trace_unit = 0
     contains
     ⟨SF base: sf chain: TBP⟩
    end type sf_chain_t
```

Finalizer.

⟨*SF base: sf chain: TBP*⟩≡
```
    procedure :: final => sf_chain_final
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_final (object)
      class(sf_chain_t), intent(inout) :: object
      integer :: i
      call object%final_tracing ()
      if (allocated (object%sf)) then
        do i = 1, size (object%sf, 1)
          associate (sf => object%sf(i))
            if (allocated (sf%int)) then
              call sf%int%final ()
            end if
          end associate
        end do
      end if
      call beam_final (object%beam_t)
    end subroutine sf_chain_final
```

Output.

⟨*SF base: sf chain: TBP*⟩+≡
```
    procedure :: write => sf_chain_write
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_write (object, unit)
      class(sf_chain_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "Incoming particles / structure-function chain:"
      if (associated (object%beam_data)) then
        write (u, "(3x,A,I0)")  "n_in    = ", object%n_in
        write (u, "(3x,A,I0)")  "n_strfun = ", object%n_strfun
```

1096

```
          write (u, "(3x,A,I0)")  "n_par     = ", object%n_par
          if (object%n_par /= object%n_bound) then
              write (u, "(3x,A,I0)")  "n_bound   = ", object%n_bound
          end if
          call beam_data_write (object%beam_data, u)
          call write_separator (u)
          call beam_write (object%beam_t, u)
          if (allocated (object%sf)) then
              do i = 1, object%n_strfun
                 associate (sf => object%sf(i))
                   call write_separator (u)
                   if (allocated (sf%int)) then
                      call sf%int%write (u)
                   else
                      write (u, "(1x,A)")  "SF instance: [undefined]"
                   end if
                 end associate
              end do
          end if
        else
           write (u, "(3x,A)")  "[undefined]"
        end if
    end subroutine sf_chain_write
```

Initialize: setup beams. The `beam_data` target must remain valid for the lifetime of the chain, since we just establish a pointer. The structure-function configuration array is used to initialize the individual structure-function entries. The target attribute is needed because the `sf_int` entries establish pointers to the configuration data.

⟨*SF base: sf chain: TBP*⟩+≡
```
    procedure :: init => sf_chain_init
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_init (sf_chain, beam_data, sf_config)
      class(sf_chain_t), intent(out) :: sf_chain
      type(beam_data_t), intent(in), target :: beam_data
      type(sf_config_t), dimension(:), intent(in), optional, target :: sf_config
      integer :: i
      sf_chain%beam_data => beam_data
      sf_chain%n_in = beam_data_get_n_in (beam_data)
      call beam_init (sf_chain%beam_t, beam_data)
      if (present (sf_config)) then
         sf_chain%n_strfun = size (sf_config)
         allocate (sf_chain%sf (sf_chain%n_strfun))
         do i = 1, sf_chain%n_strfun
            call sf_chain%set_strfun (i, sf_config(i)%i, sf_config(i)%data)
         end do
      end if
    end subroutine sf_chain_init
```

Receive the beam momenta from a source to which the beam interaction is linked.

⟨*SF base: sf chain: TBP*⟩+≡

```
      procedure :: receive_beam_momenta => sf_chain_receive_beam_momenta
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_chain_receive_beam_momenta (sf_chain)
      class(sf_chain_t), intent(inout), target :: sf_chain
      type(interaction_t), pointer :: beam_int
      beam_int => sf_chain%get_beam_int_ptr ()
      call interaction_receive_momenta (beam_int)
    end subroutine sf_chain_receive_beam_momenta
```

Explicitly set the beam momenta.

⟨SF base: sf chain: TBP⟩+≡
```
    procedure :: set_beam_momenta => sf_chain_set_beam_momenta
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_chain_set_beam_momenta (sf_chain, p)
      class(sf_chain_t), intent(inout) :: sf_chain
      type(vector4_t), dimension(:), intent(in) :: p
      call beam_set_momenta (sf_chain%beam_t, p)
    end subroutine sf_chain_set_beam_momenta
```

Set a structure-function entry. We use the `data` input to allocate the `int` structure-function instance with appropriate type, then initialize the entry. The entry establishes a pointer to `data`.

The index `i` is the structure-function index in the chain.

⟨SF base: sf chain: TBP⟩+≡
```
    procedure :: set_strfun => sf_chain_set_strfun
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_chain_set_strfun (sf_chain, i, beam_index, data)
      class(sf_chain_t), intent(inout) :: sf_chain
      integer, intent(in) :: i
      integer, dimension(:), intent(in) :: beam_index
      class(sf_data_t), intent(in), target :: data
      integer :: n_par, j
      n_par = data%get_n_par ()
      call data%allocate_sf_int (sf_chain%sf(i)%int)
      associate (sf_int => sf_chain%sf(i)%int)
        call sf_int%init (data)
        call sf_int%set_beam_index (beam_index)
        call sf_int%set_par_index &
             ([(j, j = sf_chain%n_par + 1, sf_chain%n_par + n_par)])
        sf_chain%n_par = sf_chain%n_par + n_par
        if (.not. data%is_generator ()) then
           sf_chain%n_bound = sf_chain%n_bound + n_par
        end if
      end associate
    end subroutine sf_chain_set_strfun
```

Return the number of structure-function parameters.

⟨SF base: sf chain: TBP⟩+≡
```
    procedure :: get_n_par => sf_chain_get_n_par
    procedure :: get_n_bound => sf_chain_get_n_bound
```

⟨*SF base: procedures*⟩+≡

```
function sf_chain_get_n_par (sf_chain) result (n)
  class(sf_chain_t), intent(in) :: sf_chain
  integer :: n
  n = sf_chain%n_par
end function sf_chain_get_n_par

function sf_chain_get_n_bound (sf_chain) result (n)
  class(sf_chain_t), intent(in) :: sf_chain
  integer :: n
  n = sf_chain%n_bound
end function sf_chain_get_n_bound
```

Return a pointer to the beam interaction.

⟨*SF base: sf chain: TBP*⟩+≡

```
procedure :: get_beam_int_ptr => sf_chain_get_beam_int_ptr
```

⟨*SF base: procedures*⟩+≡

```
function sf_chain_get_beam_int_ptr (sf_chain) result (int)
  class(sf_chain_t), intent(in), target :: sf_chain
  type(interaction_t), pointer :: int
  int => beam_get_int_ptr (sf_chain%beam_t)
end function sf_chain_get_beam_int_ptr
```

Enable the trace feature: record structure function data (input parameters, x values, evaluation result) to an external file.

⟨*SF base: sf chain: TBP*⟩+≡

```
procedure :: setup_tracing => sf_chain_setup_tracing
procedure :: final_tracing => sf_chain_final_tracing
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_setup_tracing (sf_chain, file)
  class(sf_chain_t), intent(inout) :: sf_chain
  type(string_t), intent(in) :: file
  if (sf_chain%n_strfun > 0) then
     sf_chain%trace_enable = .true.
     sf_chain%trace_unit = free_unit ()
     open (sf_chain%trace_unit, file = char (file), action = "write", &
           status = "replace")
     call sf_chain%write_trace_header ()
  else
     call msg_error ("Beam structure: no structure functions, tracing &
           &disabled")
  end if
end subroutine sf_chain_setup_tracing

subroutine sf_chain_final_tracing (sf_chain)
  class(sf_chain_t), intent(inout) :: sf_chain
  if (sf_chain%trace_enable) then
     close (sf_chain%trace_unit)
     sf_chain%trace_enable = .false.
  end if
end subroutine sf_chain_final_tracing
```

Write the header for the tracing file.

⟨*SF base: sf chain: TBP*⟩+≡
```
procedure :: write_trace_header => sf_chain_write_trace_header
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_write_trace_header (sf_chain)
  class(sf_chain_t), intent(in) :: sf_chain
  integer :: u
  if (sf_chain%trace_enable) then
     u = sf_chain%trace_unit
     write (u, "('# ',A)")  "WHIZARD output: &
          &structure-function sampling data"
     write (u, "('# ',A,1x,I0)")  "Number of sf records:", sf_chain%n_strfun
     write (u, "('# ',A,1x,I0)")  "Number of parameters:", sf_chain%n_par
     write (u, "('# ',A)")  "Columns: channel, p(n_par), x(n_par), f, Jac * f"
  end if
end subroutine sf_chain_write_trace_header
```

Write a record which collects the structure function data for the current data point. For the selected channel, we print first the input integration parameters, then the x values, then the structure-function value summed over all quantum numbers, then the structure function value times the mapping Jacobian.

⟨*SF base: sf chain: TBP*⟩+≡
```
procedure :: trace => sf_chain_trace
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_trace (sf_chain, c_sel, p, x, f, sf_sum)
  class(sf_chain_t), intent(in) :: sf_chain
  integer, intent(in) :: c_sel
  real(default), dimension(:,:), intent(in) :: p
  real(default), dimension(:), intent(in) :: x
  real(default), dimension(:), intent(in) :: f
  real(default), intent(in) :: sf_sum
  integer :: u, i
  if (sf_chain%trace_enable) then
     u = sf_chain%trace_unit
     write (u, "(1x,I0)", advance="no")  c_sel
     write (u, "(2x)", advance="no")
     do i = 1, sf_chain%n_par
        write (u, "(1x," // FMT_17 // ")", advance="no")  p(i,c_sel)
     end do
     write (u, "(2x)", advance="no")
     do i = 1, sf_chain%n_par
        write (u, "(1x," // FMT_17 // ")", advance="no")  x(i)
     end do
     write (u, "(2x)", advance="no")
     write (u, "(2(1x," // FMT_17 // "))")  sf_sum, f(c_sel) * sf_sum
  end if
end subroutine sf_chain_trace
```

1100

10.3.8 Chain instances

A structure-function chain instance contains copies of the interactions in the configuration chain, suitably linked to each other and connected by evaluators.

After initialization, `out_sf` should point, for each beam, to the last structure function that affects this beam. `out_sf_i` should indicate the index of the corresponding outgoing particle within that structure-function interaction.

Analogously, `out_eval` is the last evaluator in the structure-function chain, which contains the complete set of outgoing particles. `out_eval_i` should indicate the index of the outgoing particles, within that evaluator, which will initiate the collision.

When calculating actual kinematics, we fill the `p`, `r`, and `x` arrays and the `f` factor. The `p` array denotes the MC input parameters as they come from the random-number generator. The `r` array results from applying global mappings. The `x` array results from applying structure-function local mappings. The x values can be interpreted directly as momentum fractions (or angle fractions, where recoil is involved). The `f` factor is the Jacobian that results from applying all mappings.

The `mapping` entry may store a global mapping that is applied to a combination of x values and structure functions, as opposed to mappings that affect only a single structure function. It is applied before the latter mappings, in the transformation from the `p` array to the `r` array. For parameters affected by this mapping, we should ensure that they are not involved in a local mapping.

⟨*SF base: public*⟩+≡
```
    public :: sf_chain_instance_t
```

⟨*SF base: types*⟩+≡
```
    type, extends (beam_t) :: sf_chain_instance_t
      type(sf_chain_t), pointer :: config => null ()
      integer :: status = SF_UNDEFINED
      type(sf_instance_t), dimension(:), allocatable :: sf
      integer, dimension(:), allocatable :: out_sf
      integer, dimension(:), allocatable :: out_sf_i
      integer :: out_eval = 0
      integer, dimension(:), allocatable :: out_eval_i
      integer :: selected_channel = 0
      real(default), dimension(:,:), allocatable :: p, pb
      real(default), dimension(:,:), allocatable :: r, rb
      real(default), dimension(:), allocatable :: f
      real(default), dimension(:), allocatable :: x
      logical, dimension(:), allocatable :: bound
      real(default) :: x_free = 1
      type(sf_channel_t), dimension(:), allocatable :: channel
    contains
    ⟨SF base: sf chain instance: TBP⟩
    end type sf_chain_instance_t
```

Finalizer.

⟨*SF base: sf chain instance: TBP*⟩≡
```
    procedure :: final => sf_chain_instance_final
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_instance_final (object)
```

```
      class(sf_chain_instance_t), intent(inout) :: object
      integer :: i
      if (allocated (object%sf)) then
         do i = 1, size (object%sf, 1)
            associate (sf => object%sf(i))
              if (allocated (sf%int)) then
                 call evaluator_final (sf%eval)
                 call sf%int%final ()
              end if
            end associate
         end do
      end if
      call beam_final (object%beam_t)
   end subroutine sf_chain_instance_final
```

Output.

Note: nagfor 5.3.1 appears to be slightly confused with the allocation status. We check both for allocation and nonzero size.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
   procedure :: write => sf_chain_instance_write
```

⟨*SF base: procedures*⟩+≡
```
   subroutine sf_chain_instance_write (object, unit)
     class(sf_chain_instance_t), intent(in) :: object
     integer, intent(in), optional :: unit
     integer :: u, i, c
     u = output_unit (unit)
     write (u, "(1x,A)", advance="no")  "Structure-function chain instance:"
     call write_sf_status (object%status, u)
     if (allocated (object%out_sf)) then
        write (u, "(3x,A)", advance="no")  "outgoing (interactions) ="
        do i = 1, size (object%out_sf)
           write (u, "(1x,I0,':',I0)", advance="no") &
                 object%out_sf(i), object%out_sf_i(i)
        end do
        write (u, *)
     end if
     if (object%out_eval /= 0) then
        write (u, "(3x,A)", advance="no")  "outgoing (evaluators)   ="
        do i = 1, size (object%out_sf)
           write (u, "(1x,I0,':',I0)", advance="no") &
                 object%out_eval, object%out_eval_i(i)
        end do
        write (u, *)
     end if
     if (allocated (object%sf)) then
        if (size (object%sf) /= 0) then
           write (u, "(1x,A)")  "Structure-function parameters:"
           do c = 1, size (object%f)
              write (u, "(1x,A,I0,A)", advance="no")  "Channel #", c, ":"
              if (c == object%selected_channel) then
                 write (u, "(1x,A)")  "[selected]"
              else
                 write (u, *)
```

```
                end if
                write (u, "(3x,A,9(1x,F9.7))")  "p =", object%p(:,c)
                write (u, "(3x,A,9(1x,F9.7))")  "r =", object%r(:,c)
                write (u, "(3x,A,9(1x,ES13.7))")  "f =", object%f(c)
                write (u, "(3x,A)", advance="no") "m ="
                call object%channel(c)%write (u)
            end do
            write (u, "(3x,A,9(1x,F9.7))")  "x =", object%x
            if (.not. all (object%bound)) then
                write (u, "(3x,A,9(1x,L1))")  "bound =", object%bound
            end if
        end if
    end if
    call write_separator (u)
    call beam_write (object%beam_t, u)
    if (allocated (object%sf)) then
        do i = 1, size (object%sf)
            associate (sf => object%sf(i))
                call write_separator (u)
                if (allocated (sf%int)) then
                    if (allocated (sf%r)) then
                        write (u, "(1x,A)")  "Structure-function parameters:"
                        do c = 1, size (sf%f)
                            write (u, "(1x,A,I0,A)", advance="no")  "Channel #", c, ":"
                            if (c == object%selected_channel) then
                                write (u, "(1x,A)")  "[selected]"
                            else
                                write (u, *)
                            end if
                            write (u, "(3x,A,9(1x,F9.7))")  "r =", sf%r(:,c)
                            write (u, "(3x,A,9(1x,ES13.7))")  "f =", sf%f(c)
                            write (u, "(3x,A,9(1x,L1,7x))") "m =", sf%m(c)
                        end do
                        write (u, "(3x,A,9(1x,F9.7))")  "x =", sf%x
                    end if
                    call sf%int%write (u)
                    if (.not. evaluator_is_empty (sf%eval)) then
                        call sf%eval%write (u)
                    end if
                end if
            end associate
        end do
    end if
  end subroutine sf_chain_instance_write
```

Initialize. This creates a copy of the interactions in the configuration chain, assumed to be properly initialized. In the copy, we allocate the p etc. arrays.

The brute-force assignment of the sf component would be straightforward, but at least gfortran 4.6.3 would like a more fine-grained copy. In any case, the copy is deep as far as allocatables are concerned, but for the contained interaction_t objects the copy is shallow, as long as we do not bind defined assignment to the type. Therefore, we have to re-assign the interaction_t components explicitly, this time calling the proper defined assignment. Fur-

thermore, we allocate the parameter arrays for each structure function.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: init => sf_chain_instance_init
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_instance_init (chain, config, n_channel)
  class(sf_chain_instance_t), intent(out), target :: chain
  type(sf_chain_t), intent(in), target :: config
  integer, intent(in) :: n_channel
  integer :: i, j
  integer :: n_par_tot, n_par, n_strfun
  chain%config => config
  n_strfun = config%n_strfun
  chain%beam_t = config%beam_t
  allocate (chain%out_sf (config%n_in), chain%out_sf_i (config%n_in))
  allocate (chain%out_eval_i (config%n_in))
  chain%out_sf = 0
  chain%out_sf_i = [(i, i = 1, config%n_in)]
  chain%out_eval_i = chain%out_sf_i
  n_par_tot = 0
  if (n_strfun /= 0) then
     allocate (chain%sf (n_strfun))
     do i = 1, n_strfun
        associate (sf => chain%sf(i))
          allocate (sf%int, source=config%sf(i)%int)
          sf%int%interaction_t = config%sf(i)%int%interaction_t
          n_par = size (sf%int%par_index)
          allocate (sf%r (n_par, n_channel));  sf%r = 0
          allocate (sf%rb(n_par, n_channel));  sf%rb= 0
          allocate (sf%f (n_channel));         sf%f = 0
          allocate (sf%m (n_channel));         sf%m = .false.
          allocate (sf%x (n_par));             sf%x = 0
          n_par_tot = n_par_tot + n_par
        end associate
     end do
     allocate (chain%p (n_par_tot, n_channel));  chain%p = 0
     allocate (chain%pb(n_par_tot, n_channel));  chain%pb= 0
     allocate (chain%r (n_par_tot, n_channel));  chain%r = 0
     allocate (chain%rb(n_par_tot, n_channel));  chain%rb= 0
     allocate (chain%f (n_channel));             chain%f = 0
     allocate (chain%x (n_par_tot));             chain%x = 0
     call allocate_sf_channels &
          (chain%channel, n_channel=n_channel, n_strfun=n_strfun)
  end if
  allocate (chain%bound (n_par_tot), source = .true.)
  do i = 1, n_strfun
     associate (sf => chain%sf(i))
       if (sf%int%is_generator ()) then
          do j = 1, size (sf%int%par_index)
             chain%bound(sf%int%par_index(j)) = .false.
          end do
       end if
     end associate
  end do
  chain%status = SF_INITIAL
```

```
      end subroutine sf_chain_instance_init
```

Manually select a channel.

⟨SF base: sf chain instance: TBP⟩+≡
```
    procedure :: select_channel => sf_chain_instance_select_channel
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_chain_instance_select_channel (chain, channel)
      class(sf_chain_instance_t), intent(inout) :: chain
      integer, intent(in), optional :: channel
      if (present (channel)) then
         chain%selected_channel = channel
      else
         chain%selected_channel = 0
      end if
    end subroutine sf_chain_instance_select_channel
```

Copy a channel-mapping object to the structure-function chain instance. We
assume that assignment is sufficient, i.e., any non-static components of the
`channel` object are allocatable und thus recursively copied.

   After the copy, we extract the single-entry mappings and activate them for
the individual structure functions. If there is a multi-entry mapping, we obtain
the corresponding MC parameter indices and set them in the copy of the channel
object.

⟨SF base: sf chain instance: TBP⟩+≡
```
    procedure :: set_channel => sf_chain_instance_set_channel
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_chain_instance_set_channel (chain, c, channel)
      class(sf_chain_instance_t), intent(inout) :: chain
      integer, intent(in) :: c
      type(sf_channel_t), intent(in) :: channel
      integer :: i, j, k
      if (chain%status >= SF_INITIAL) then
         chain%channel(c) = channel
         j = 0
         do i = 1, chain%config%n_strfun
            associate (sf => chain%sf(i))
              sf%m(c) = channel%is_single_mapping (i)
              if (channel%is_multi_mapping (i)) then
                 do k = 1, size (sf%int%beam_index)
                    j = j + 1
                    call chain%channel(c)%set_par_index (j, sf%int%par_index(k))
                 end do
              end if
            end associate
         end do
         chain%status = SF_INITIAL
      end if
    end subroutine sf_chain_instance_set_channel
```

Link the interactions in the chain. First, link the beam instance to its template in the configuration chain, which should have the appropriate momenta fixed.

Then, we follow the chain via the arrays `out_sf` and `out_sf_i`. The arrays are (up to) two-dimensional, the entries correspond to the beam particle(s). For each beam, the entry `out_sf` points to the last interaction that affected this beam, and `out_sf_i` is the out-particle index within that interaction. For the initial beam, `out_sf` is zero by definition.

For each entry in the chain, we scan the affected beams (one or two). We look for `out_sf` and link the out-particle there to the corresponding in-particle in the current interaction. Then, we update the entry in `out_sf` and `out_sf_i` to point to the current interaction.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
  procedure :: link_interactions => sf_chain_instance_link_interactions
```

⟨*SF base: procedures*⟩+≡
```
  subroutine sf_chain_instance_link_interactions (chain)
    class(sf_chain_instance_t), intent(inout), target :: chain
    type(interaction_t), pointer :: int
    integer :: i, j, b
    if (chain%status >= SF_INITIAL) then
       do b = 1, chain%config%n_in
          int => beam_get_int_ptr (chain%beam_t)
          call interaction_set_source_link (int, b, &
               chain%config%beam_t, b)
       end do
       if (allocated (chain%sf)) then
          do i = 1, size (chain%sf)
             associate (sf_int => chain%sf(i)%int)
               do j = 1, size (sf_int%beam_index)
                  b = sf_int%beam_index(j)
                  call link (sf_int%interaction_t, b, sf_int%incoming(j))
                  chain%out_sf(b) = i
                  chain%out_sf_i(b) = sf_int%outgoing(j)
               end do
             end associate
          end do
       end if
       chain%status = SF_DONE_LINKS
    end if
  contains
    subroutine link (int, b, in_index)
      type(interaction_t), intent(inout) :: int
      integer, intent(in) :: b, in_index
      integer :: i
      i = chain%out_sf(b)
      select case (i)
      case (0)
         call interaction_set_source_link (int, in_index, &
              chain%beam_t, chain%out_sf_i(b))
      case default
         call interaction_set_source_link (int, in_index, &
              chain%sf(i)%int%interaction_t, chain%out_sf_i(b))
      end select
    end subroutine link
```

1106

```
          end subroutine sf_chain_instance_link_interactions
```

Exchange the quantum-number masks between the interactions in the chain, so we can combine redundant entries and detect any obvious mismatch.

We proceed first in the forward direction and then backwards again.

After this is finished, we finalize initialization by calling the `setup_constants` method, which prepares constant data that depend on the matrix element structure.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: exchange_mask => sf_chain_exchange_mask
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_exchange_mask (chain)
      class(sf_chain_instance_t), intent(inout), target :: chain
      type(interaction_t), pointer :: int
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
      integer :: i
      if (chain%status >= SF_DONE_LINKS) then
         if (allocated (chain%sf)) then
            int => beam_get_int_ptr (chain%beam_t)
            allocate (mask (interaction_get_n_out (int)))
            mask = interaction_get_mask (int)
            if (size (chain%sf) /= 0) then
               do i = 1, size (chain%sf) - 1
                  call interaction_exchange_mask (chain%sf(i)%int%interaction_t)
               end do
               do i = size (chain%sf), 1, -1
                  call interaction_exchange_mask (chain%sf(i)%int%interaction_t)
               end do
               if (any (mask .neqv. interaction_get_mask (int))) then
                  chain%status = SF_FAILED_MASK
                  return
               end if
               do i = 1, size (chain%sf)
                  call chain%sf(i)%int%setup_constants ()
               end do
            end if
         end if
         chain%status = SF_DONE_MASK
      end if
    end subroutine sf_chain_exchange_mask
```

Initialize the evaluators that connect the interactions in the chain.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: init_evaluators => sf_chain_instance_init_evaluators
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_instance_init_evaluators (chain)
      class(sf_chain_instance_t), intent(inout), target :: chain
      type(interaction_t), pointer :: int
      type(quantum_numbers_mask_t) :: mask
      integer :: i
      if (chain%status >= SF_DONE_MASK) then
```

1107

```
            if (allocated (chain%sf)) then
                if (size (chain%sf) /= 0) then
                    mask = new_quantum_numbers_mask (.false., .false., .true.)
                    int => beam_get_int_ptr (chain%beam_t)
                    do i = 1, size (chain%sf)
                        associate (sf => chain%sf(i))
                            call evaluator_init_product (sf%eval, &
                                int, sf%int%interaction_t, &
                                mask)
                            if (evaluator_is_empty (sf%eval)) then
                                chain%status = SF_FAILED_CONNECTIONS
                                return
                            end if
                            int => evaluator_get_int_ptr (sf%eval)
                        end associate
                    end do
                    call find_outgoing_particles ()
                end if
            end if
            chain%status = SF_DONE_CONNECTIONS
        end if
    contains
        ⟨SF base: init evaluators: find outgoing particles⟩
    end subroutine sf_chain_instance_init_evaluators
```

This is an internal subroutine of the previous one: After evaluators are set, trace the outgoing particles to the last evaluator. We only need the first channel, all channels are equivalent for this purpose.

For each beam, the outgoing particle is located by `out_sf` (the structure-function object where it originates) and `out_sf_i` (the index within that object). This particle is referenced by the corresponding evaluator, which in turn is referenced by the next evaluator, until we are at the end of the chain. We can trace back references by `interaction_find_link`. Knowing that `out_eval` is the index of the last evaluator, we thus determine `out_eval_i`, the index of the outgoing particle within that evaluator.

⟨SF base: init evaluators: find outgoing particles⟩≡
```
    subroutine find_outgoing_particles ()
        type(interaction_t), pointer :: int, int_next
        integer :: i, j, out_sf, out_i
        chain%out_eval = size (chain%sf)
        do j = 1, size (chain%out_eval_i)
            out_sf = chain%out_sf(j)
            out_i = chain%out_sf_i(j)
            if (out_sf == 0) then
                int => beam_get_int_ptr (chain%beam_t)
                out_sf = 1
            else
                int => chain%sf(out_sf)%int%interaction_t
            end if
            do i = out_sf, chain%out_eval
                int_next => evaluator_get_int_ptr (chain%sf(i)%eval)
                out_i = interaction_find_link (int_next, int, out_i)
                int => int_next
```

```
            end do
         chain%out_eval_i(j) = out_i
      end do
   end subroutine find_outgoing_particles
```

Compute the kinematics in the chain instance. We can assume that the seed momenta are set in the configuration beams. Scanning the chain, we first transfer the incoming momenta. Then, the use up the MC input parameter array `p` to compute the radiated and outgoing momenta.

In the multi-channel case, `c_sel` is the channel which we use for computing the kinematics and the `x` values. In the other channels, we invert the kinematics in order to recover the corresponding rows in the `r` array, and the Jacobian `f`.

We first apply any global mapping to transform the input `p` into the array `r`. This is then given to the structure functions which compute the final array `x` and Jacobian factors `f`, which we multiply to obtain the overall Jacobian.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
      procedure :: compute_kinematics => sf_chain_instance_compute_kinematics
```

⟨*SF base: procedures*⟩+≡
```
   subroutine sf_chain_instance_compute_kinematics (chain, c_sel, p_in)
     class(sf_chain_instance_t), intent(inout), target :: chain
     integer, intent(in) :: c_sel
     real(default), dimension(:), intent(in) :: p_in
     type(interaction_t), pointer :: int
     real(default) :: f_mapping
     logical, dimension(size (chain%bound)) :: bound
     integer :: i, j, c
     if (chain%status >= SF_DONE_CONNECTIONS) then
         call chain%select_channel (c_sel)
         int => beam_get_int_ptr (chain%beam_t)
         call interaction_receive_momenta (int)
         if (allocated (chain%sf)) then
            if (size (chain%sf) /= 0) then
               forall (i = 1:size (chain%sf))  chain%sf(i)%int%status = SF_INITIAL
               !!! Bug in nagfor 5.3.1(907), fixed in 5.3.1(982)
               ! chain%p (:,c_sel) = unpack (p_in, chain%bound, 0._default)
               !!! Workaround:
               bound = chain%bound
               chain%p (:,c_sel) = unpack (p_in, bound, 0._default)
               chain%pb(:,c_sel) = 1 - chain%p(:,c_sel)
               chain%f = 1
               chain%x_free = 1
               do i = 1, size (chain%sf)
                  associate (sf => chain%sf(i))
                    call sf%int%generate_free (sf%r(:,c_sel), sf%rb(:,c_sel), &
                         chain%x_free)
                    do j = 1, size (sf%x)
                       if (.not. chain%bound(sf%int%par_index(j))) then
                          chain%p (sf%int%par_index(j),c_sel) = sf%r (j,c_sel)
                          chain%pb(sf%int%par_index(j),c_sel) = sf%rb(j,c_sel)
                       end if
                    end do
                  end associate
               end do
```

```fortran
if (allocated (chain%channel(c_sel)%multi_mapping)) then
   call chain%channel(c_sel)%multi_mapping%compute &
        (chain%r(:,c_sel), chain%rb(:,c_sel), &
         f_mapping, &
         chain%p(:,c_sel), chain%pb(:,c_sel), &
         chain%x_free)
   chain%f(c_sel) = f_mapping
else
   chain%r (:,c_sel) = chain%p (:,c_sel)
   chain%rb(:,c_sel) = chain%pb(:,c_sel)
   chain%f(c_sel) = 1
end if
do i = 1, size (chain%sf)
   associate (sf => chain%sf(i))
     call sf%int%seed_kinematics ()
     do j = 1, size (sf%x)
        sf%r (j,c_sel) = chain%r (sf%int%par_index(j),c_sel)
        sf%rb(j,c_sel) = chain%rb(sf%int%par_index(j),c_sel)
     end do
     call sf%int%complete_kinematics &
          (sf%x, sf%f(c_sel), sf%r(:,c_sel), sf%rb(:,c_sel), &
           sf%m(c_sel))
     do j = 1, size (sf%x)
        chain%x(sf%int%par_index(j)) = sf%x(j)
     end do
     if (sf%int%status <= SF_FAILED_KINEMATICS) then
        chain%status = SF_FAILED_KINEMATICS
        return
     end if
     do c = 1, size (sf%f)
        if (c /= c_sel) then
           call sf%int%inverse_kinematics &
                (sf%x, sf%f(c), sf%r(:,c), sf%rb(:,c), sf%m(c))
           do j = 1, size (sf%x)
              chain%r (sf%int%par_index(j),c) = sf%r (j,c)
              chain%rb(sf%int%par_index(j),c) = sf%rb(j,c)
           end do
        end if
        chain%f(c) = chain%f(c) * sf%f(c)
     end do
     if (.not. evaluator_is_empty (sf%eval)) then
        call evaluator_receive_momenta (sf%eval)
     end if
   end associate
end do
do c = 1, size (chain%f)
   if (c /= c_sel) then
      if (allocated (chain%channel(c)%multi_mapping)) then
         call chain%channel(c)%multi_mapping%inverse &
              (chain%r(:,c), chain%rb(:,c), &
               f_mapping, &
               chain%p(:,c), chain%pb(:,c), &
               chain%x_free)
         chain%f(c) = chain%f(c) * f_mapping
```

```
                  else
                      chain%p (:,c) = chain%r (:,c)
                      chain%pb(:,c) = chain%rb(:,c)
                  end if
              end if
          end do
      end if
  end if
  chain%status = SF_DONE_KINEMATICS
      end if
  end subroutine sf_chain_instance_compute_kinematics
```

This is a variant of the previous procedure. We know the x parameters and reconstruct the momenta and the MC input parameters p. We do not need to select a channel.

Note: this is probably redundant, since the method we actually want starts from the momenta, recovers all x parameters, and then inverts mappings. See below `recover_kinematics`.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: inverse_kinematics => sf_chain_instance_inverse_kinematics
```

⟨*SF base: procedures*⟩+≡
```
  subroutine sf_chain_instance_inverse_kinematics (chain, x)
    class(sf_chain_instance_t), intent(inout), target :: chain
    real(default), dimension(:), intent(in) :: x
    type(interaction_t), pointer :: int
    real(default) :: f_mapping
    integer :: i, j, c
    if (chain%status >= SF_DONE_CONNECTIONS) then
        call chain%select_channel ()
        int => beam_get_int_ptr (chain%beam_t)
        call interaction_receive_momenta (int)
        if (allocated (chain%sf)) then
            chain%f = 1
            if (size (chain%sf) /= 0) then
                forall (i = 1:size (chain%sf))  chain%sf(i)%int%status = SF_INITIAL
                chain%x = x
                do i = 1, size (chain%sf)
                    associate (sf => chain%sf(i))
                      call sf%int%seed_kinematics ()
                      do j = 1, size (sf%x)
                          sf%x(j) = chain%x(sf%int%par_index(j))
                      end do
                      do c = 1, size (sf%f)
                          call sf%int%inverse_kinematics &
                                (sf%x, sf%f(c), sf%r(:,c), sf%rb(:,c), sf%m(c), c==1)
                          chain%f(c) = chain%f(c) * sf%f(c)
                          do j = 1, size (sf%x)
                              chain%r (sf%int%par_index(j),c) = sf%r (j,c)
                              chain%rb(sf%int%par_index(j),c) = sf%rb(j,c)
                          end do
                      end do
                      if (.not. evaluator_is_empty (sf%eval)) then
                          call evaluator_receive_momenta (sf%eval)
```

```
                  end if
               end associate
            end do
            do c = 1, size (chain%f)
               if (allocated (chain%channel(c)%multi_mapping)) then
                  call chain%channel(c)%multi_mapping%inverse &
                       (chain%r(:,c), chain%rb(:,c), &
                       f_mapping, &
                       chain%p(:,c), chain%pb(:,c), &
                       chain%x_free)
                  chain%f(c) = chain%f(c) * f_mapping
               else
                  chain%p (:,c) = chain%r (:,c)
                  chain%pb(:,c) = chain%rb(:,c)
               end if
            end do
         end if
      end if
      chain%status = SF_DONE_KINEMATICS
   end if
 end subroutine sf_chain_instance_inverse_kinematics
```

Recover the kinematics: assuming that the last evaluator has been filled with a valid set of momenta, we travel the momentum links backwards and fill the preceding evaluators and, as a side effect, interactions. We stop at the beam interaction.

After all momenta are set, apply the `inverse_kinematics` procedure above, suitably modified, to recover the x and p parameters and the Jacobian factors.

The `c_sel` (channel) argument is just used to mark a selected channel for the records, otherwise the recovery procedure is independent of this.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: recover_kinematics => sf_chain_instance_recover_kinematics
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_instance_recover_kinematics (chain, c_sel)
      class(sf_chain_instance_t), intent(inout), target :: chain
      integer, intent(in) :: c_sel
      real(default) :: f_mapping
      integer :: i, j, c
      if (chain%status >= SF_DONE_CONNECTIONS) then
         call chain%select_channel (c_sel)
         if (allocated (chain%sf)) then
            do i = size (chain%sf), 1, -1
               associate (sf => chain%sf(i))
                 if (.not. evaluator_is_empty (sf%eval)) then
                    call evaluator_send_momenta (sf%eval)
                 end if
               end associate
            end do
            chain%f = 1
            if (size (chain%sf) /= 0) then
               forall (i = 1:size (chain%sf))  chain%sf(i)%int%status = SF_INITIAL
               chain%x_free = 1
```

```
              do i = 1, size (chain%sf)
                 associate (sf => chain%sf(i))
                    call sf%int%seed_kinematics ()
                    call sf%int%recover_x (sf%x, chain%x_free)
                    do j = 1, size (sf%x)
                       chain%x(sf%int%par_index(j)) = sf%x(j)
                    end do
                    do c = 1, size (sf%f)
                       call sf%int%inverse_kinematics &
                            (sf%x, sf%f(c), sf%r(:,c), sf%rb(:,c), sf%m(c), c==1)
                       chain%f(c) = chain%f(c) * sf%f(c)
                       do j = 1, size (sf%x)
                          chain%r (sf%int%par_index(j),c) = sf%r (j,c)
                          chain%rb(sf%int%par_index(j),c) = sf%rb(j,c)
                       end do
                    end do
                 end associate
              end do
              do c = 1, size (chain%f)
                 if (allocated (chain%channel(c)%multi_mapping)) then
                    call chain%channel(c)%multi_mapping%inverse &
                         (chain%r(:,c), chain%rb(:,c), &
                         f_mapping, &
                         chain%p(:,c), chain%pb(:,c), &
                         chain%x_free)
                    chain%f(c) = chain%f(c) * f_mapping
                 else
                    chain%p (:,c) = chain%r (:,c)
                    chain%pb(:,c) = chain%rb(:,c)
                 end if
              end do
           end if
        end if
        chain%status = SF_DONE_KINEMATICS
     end if
   end subroutine sf_chain_instance_recover_kinematics
```

Return the initial beam momenta to their source, thus completing kinematics recovery. Obviously, this works as a side effect.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: return_beam_momenta => sf_chain_instance_return_beam_momenta
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_chain_instance_return_beam_momenta (chain)
      class(sf_chain_instance_t), intent(in), target :: chain
      type(interaction_t), pointer :: int
      if (chain%status >= SF_DONE_KINEMATICS) then
         int => beam_get_int_ptr (chain%beam_t)
         call interaction_send_momenta (int)
      end if
    end subroutine sf_chain_instance_return_beam_momenta
```

Evaluate all interactions in the chain and the product evaluators. We provide

a `scale` argument that is given to all structure functions in the chain.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: evaluate => sf_chain_instance_evaluate
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_instance_evaluate (chain, scale)
  class(sf_chain_instance_t), intent(inout), target :: chain
  real(default), intent(in) :: scale
  type(interaction_t), pointer :: out_int
  real(default) :: sf_sum
  integer :: i
  if (chain%status >= SF_DONE_KINEMATICS) then
     if (allocated (chain%sf)) then
        if (size (chain%sf) /= 0) then
           do i = 1, size (chain%sf)
              associate (sf => chain%sf(i))
                call sf%int%apply (scale)
                if (sf%int%status <= SF_FAILED_EVALUATION) then
                   chain%status = SF_FAILED_EVALUATION
                   return
                end if
                if (.not. evaluator_is_empty (sf%eval)) then
                   call sf%eval%evaluate ()
                end if
              end associate
           end do
           out_int => chain%get_out_int_ptr ()
           sf_sum = interaction_sum (out_int)
           call chain%config%trace &
                (chain%selected_channel, chain%p, chain%x, chain%f, sf_sum)
        end if
     end if
     chain%status = SF_EVALUATED
  end if
end subroutine sf_chain_instance_evaluate
```

10.3.9 Access to the chain instance

Transfer the outgoing momenta to the array **p**. We assume that array sizes match.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_out_momenta => sf_chain_instance_get_out_momenta
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_instance_get_out_momenta (chain, p)
  class(sf_chain_instance_t), intent(in), target :: chain
  type(vector4_t), dimension(:), intent(out) :: p
  type(interaction_t), pointer :: int
  integer :: i, j
  if (chain%status >= SF_DONE_KINEMATICS) then
     do j = 1, size (chain%out_sf)
        i = chain%out_sf(j)
        select case (i)
```

```
            case (0)
               int => beam_get_int_ptr (chain%beam_t)
            case default
               int => chain%sf(i)%int%interaction_t
            end select
            p(j) = interaction_get_momentum (int, chain%out_sf_i(j))
         end do
      end if
    end subroutine sf_chain_instance_get_out_momenta
```

Return a pointer to the last evaluator in the chain (to the interaction).

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: get_out_int_ptr => sf_chain_instance_get_out_int_ptr
```

⟨*SF base: procedures*⟩+≡
```
    function sf_chain_instance_get_out_int_ptr (chain) result (int)
      class(sf_chain_instance_t), intent(in), target :: chain
      type(interaction_t), pointer :: int
      if (chain%out_eval == 0) then
         int => beam_get_int_ptr (chain%beam_t)
      else
         int => evaluator_get_int_ptr (chain%sf(chain%out_eval)%eval)
      end if
    end function sf_chain_instance_get_out_int_ptr
```

Return the index of the j-th outgoing particle, within the last evaluator.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: get_out_i => sf_chain_instance_get_out_i
```

⟨*SF base: procedures*⟩+≡
```
    function sf_chain_instance_get_out_i (chain, j) result (i)
      class(sf_chain_instance_t), intent(in) :: chain
      integer, intent(in) :: j
      integer :: i
      i = chain%out_eval_i(j)
    end function sf_chain_instance_get_out_i
```

Return the mask for the outgoing particle(s), within the last evaluator.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
    procedure :: get_out_mask => sf_chain_instance_get_out_mask
```

⟨*SF base: procedures*⟩+≡
```
    function sf_chain_instance_get_out_mask (chain) result (mask)
      class(sf_chain_instance_t), intent(in), target :: chain
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
      type(interaction_t), pointer :: int
      allocate (mask (chain%config%n_in))
      int => chain%get_out_int_ptr ()
      mask = interaction_get_mask (int, chain%out_eval_i)
    end function sf_chain_instance_get_out_mask
```

1115

Return the array of MC input parameters that corresponds to channel `c`. This is the `p` array, the parameters before all mappings.

The `p` array may be deallocated. This should correspond to a zero-size `r` argument, so nothing to do then.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_mcpar => sf_chain_instance_get_mcpar
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_instance_get_mcpar (chain, c, r)
  class(sf_chain_instance_t), intent(in) :: chain
  integer, intent(in) :: c
  real(default), dimension(:), intent(out) :: r
  if (allocated (chain%p))  r = pack (chain%p(:,c), chain%bound)
end subroutine sf_chain_instance_get_mcpar
```

Return the Jacobian factor that corresponds to channel `c`.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_f => sf_chain_instance_get_f
```

⟨*SF base: procedures*⟩+≡
```
function sf_chain_instance_get_f (chain, c) result (f)
  class(sf_chain_instance_t), intent(in) :: chain
  integer, intent(in) :: c
  real(default) :: f
  if (allocated (chain%f)) then
     f = chain%f(c)
  else
     f = 1
  end if
end function sf_chain_instance_get_f
```

Return the evaluation status.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_status => sf_chain_instance_get_status
```

⟨*SF base: procedures*⟩+≡
```
function sf_chain_instance_get_status (chain) result (status)
  class(sf_chain_instance_t), intent(in) :: chain
  integer :: status
  status = chain%status
end function sf_chain_instance_get_status
```

10.3.10 Unit tests

⟨*SF base: public*⟩+≡
```
public :: sf_base_test
```

⟨*SF base: tests*⟩≡
```
subroutine sf_base_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*SF base: execute tests*⟩
```
end subroutine sf_base_test
```

10.3.11 Test implementation: structure function

This is a template for the actual structure-function implementation which will be defined in separate modules.

Configuration data

The test structure function uses the `Test` model. It describes a scalar within an arbitrary initial particle, which is given in the initialization. The radiated particle is also a scalar, the same one, but we set its mass artificially to zero.

⟨*SF base: public*⟩+≡
```
    public :: sf_test_data_t
```

⟨*SF base: test types*⟩≡
```
    type, extends (sf_data_t) :: sf_test_data_t
      type(model_t), pointer :: model => null ()
      integer :: mode = 0
      type(flavor_t) :: flv_in
      type(flavor_t) :: flv_out
      type(flavor_t) :: flv_rad
      real(default) :: m = 0
      logical :: collinear = .true.
      real(default), dimension(:), allocatable :: qbounds
    contains
    ⟨SF base: sf test data: TBP⟩
    end type sf_test_data_t
```

Output.

⟨*SF base: sf test data: TBP*⟩≡
```
    procedure :: write => sf_test_data_write
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_data_write (data, unit, verbose)
      class(sf_test_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "SF test data:"
      write (u, "(3x,A,A)") "model    = ", char (data%model%get_name ())
      write (u, "(3x,A)", advance="no") "incoming = "
      call flavor_write (data%flv_in, u);  write (u, *)
      write (u, "(3x,A)", advance="no") "outgoing = "
      call flavor_write (data%flv_out, u);  write (u, *)
      write (u, "(3x,A)", advance="no") "radiated = "
      call flavor_write (data%flv_rad, u);  write (u, *)
      write (u, "(3x,A," // FMT_19 // ")")  "mass     = ", data%m
      write (u, "(3x,A,L1)")  "collinear = ", data%collinear
      if (.not. data%collinear .and. allocated (data%qbounds)) then
          write (u, "(3x,A," // FMT_19 // ")")  "qmin     = ", data%qbounds(1)
          write (u, "(3x,A," // FMT_19 // ")")  "qmax     = ", data%qbounds(2)
      end if
    end subroutine sf_test_data_write
```

Initialization.

⟨SF base: sf test data: TBP⟩+≡
```
    procedure :: init => sf_test_data_init
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_test_data_init (data, model, pdg_in, collinear, qbounds, mode)
      class(sf_test_data_t), intent(out) :: data
      type(model_t), intent(in), target :: model
      type(pdg_array_t), intent(in) :: pdg_in
      logical, intent(in), optional :: collinear
      real(default), dimension(2), intent(in), optional :: qbounds
      integer, intent(in), optional :: mode
      data%model => model
      if (present (mode))  data%mode = mode
      if (pdg_array_get (pdg_in, 1) /= 25) then
         call msg_fatal ("Test spectrum function: input flavor must be 's'")
      end if
      call flavor_init (data%flv_in, 25, model)
      data%m = flavor_get_mass (data%flv_in)
      if (present (collinear))  data%collinear = collinear
      call flavor_init (data%flv_out, 25, model)
      call flavor_init (data%flv_rad, 25, model)
      if (present (qbounds)) then
         allocate (data%qbounds (2))
         data%qbounds = qbounds
      end if
    end subroutine sf_test_data_init
```

Return the number of parameters: 1 if only consider collinear splitting, 3 otherwise.

⟨SF base: sf test data: TBP⟩+≡
```
    procedure :: get_n_par => sf_test_data_get_n_par
```

⟨SF base: procedures⟩+≡
```
    function sf_test_data_get_n_par (data) result (n)
      class(sf_test_data_t), intent(in) :: data
      integer :: n
      if (data%collinear) then
         n = 1
      else
         n = 3
      end if
    end function sf_test_data_get_n_par
```

Return the outgoing particle PDG code: 25

⟨SF base: sf test data: TBP⟩+≡
```
    procedure :: get_pdg_out => sf_test_data_get_pdg_out
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_test_data_get_pdg_out (data, pdg_out)
      class(sf_test_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      pdg_out(1) = 25
    end subroutine sf_test_data_get_pdg_out
```

1118

Allocate the matching interaction.

⟨*SF base: sf test data: TBP*⟩+≡
```
procedure :: allocate_sf_int => sf_test_data_allocate_sf_int
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_data_allocate_sf_int (data, sf_int)
  class(sf_test_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (sf_test_t :: sf_int)
end subroutine sf_test_data_allocate_sf_int
```

Interaction

⟨*SF base: test types*⟩+≡
```
type, extends (sf_int_t) :: sf_test_t
  type(sf_test_data_t), pointer :: data => null ()
  real(default) :: x = 0
 contains
 ⟨SF base: sf test int: TBP⟩
end type sf_test_t
```

Type string: constant

⟨*SF base: sf test int: TBP*⟩≡
```
procedure :: type_string => sf_test_type_string
```

⟨*SF base: procedures*⟩+≡
```
function sf_test_type_string (object) result (string)
  class(sf_test_t), intent(in) :: object
  type(string_t) :: string
  string = "Test"
end function sf_test_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF base: sf test int: TBP*⟩+≡
```
procedure :: write => sf_test_write
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_write (object, unit, testflag)
  class(sf_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "SF test data: [undefined]"
  end if
end subroutine sf_test_write
```

Initialize. We know that `data` will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

⟨*SF base: sf test int: TBP*⟩+≡
```
procedure :: init => sf_test_init
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_init (sf_int, data)
  class(sf_test_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  type(helicity_t) :: hel0
  type(quantum_numbers_t), dimension(3) :: qn
  mask = new_quantum_numbers_mask (.false., .false., .false.)
  select type (data)
  type is (sf_test_data_t)
     if (allocated (data%qbounds)) then
        call sf_int%base_init (mask, &
             [data%m**2], [0._default], [data%m**2], &
             [data%qbounds(1)], [data%qbounds(2)])
     else
        call sf_int%base_init (mask, &
             [data%m**2], [0._default], [data%m**2])
     end if
     sf_int%data => data
     call helicity_init (hel0, 0)
     call quantum_numbers_init (qn(1), data%flv_in,  hel0)
     call quantum_numbers_init (qn(2), data%flv_rad, hel0)
     call quantum_numbers_init (qn(3), data%flv_out, hel0)
     call interaction_add_state (sf_int%interaction_t, qn)
     call interaction_freeze (sf_int%interaction_t)
     call sf_int%set_incoming ([1])
     call sf_int%set_radiated ([2])
     call sf_int%set_outgoing ([3])
  end select
  sf_int%status = SF_INITIAL
end subroutine sf_test_init
```

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF base: sf test int: TBP*⟩+≡
```
procedure :: complete_kinematics => sf_test_complete_kinematics
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_complete_kinematics (sf_int, x, f, r, rb, map)
  class(sf_test_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
```

```
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      real(default) :: xb1
      if (map) then
         x(1) = r(1)**2
         f = 2 * r(1)
      else
         x(1) = r(1)
         f = 1
      end if
      xb1 = 1 - x(1)
      if (size (x) == 3)  x(2:3) = r(2:3)
      call sf_int%split_momentum (x, xb1)
      sf_int%x = x(1)
      select case (sf_int%status)
      case (SF_FAILED_KINEMATICS);  f = 0
      end select
    end subroutine sf_test_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF base: sf test int: TBP*⟩+≡
```
    procedure :: inverse_kinematics => sf_test_inverse_kinematics
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
      class(sf_test_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      real(default) :: xb1
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      if (map) then
         r(1) = sqrt (x(1))
         f = 2 * r(1)
      else
         r(1) = x(1)
         f = 1
      end if
      xb1 = 1 - x(1)
      if (size (x) == 3)  r(2:3) = x(2:3)
      rb = 1 - r
      sf_int%x = x(1)
      if (set_mom) then
         call sf_int%split_momentum (x, xb1)
         select case (sf_int%status)
         case (SF_FAILED_KINEMATICS);  f = 0
         end select
      end if
```

```
      end subroutine sf_test_inverse_kinematics
```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

If the `mode` indicator is one, the matrix element is equal to the parameter x.

⟨*SF base: sf test int: TBP*⟩+≡
```
   procedure :: apply => sf_test_apply
```

⟨*SF base: procedures*⟩+≡
```
   subroutine sf_test_apply (sf_int, scale)
     class(sf_test_t), intent(inout) :: sf_int
     real(default), intent(in) :: scale
     select case (sf_int%data%mode)
     case (0)
        call interaction_set_matrix_element (sf_int%interaction_t, &
             cmplx (1._default, kind=default))
     case (1)
        call interaction_set_matrix_element (sf_int%interaction_t, &
             cmplx (sf_int%x, kind=default))
     end select
     sf_int%status = SF_EVALUATED
   end subroutine sf_test_apply
```

10.3.12   Test implementation: pair spectrum

Another template, this time for a incoming particle pair, splitting into two radiated and two outgoing particles.

Configuration data

For simplicity, the spectrum contains two mirror images of the previous structure-function configuration: the incoming and all outgoing particles are test scalars.

We have two versions, one with radiated particles, one without.

⟨*SF base: test types*⟩+≡
```
   type, extends (sf_data_t) :: sf_test_spectrum_data_t
     type(model_t), pointer :: model => null ()
     type(flavor_t) :: flv_in
     type(flavor_t) :: flv_out
     type(flavor_t) :: flv_rad
     logical :: with_radiation = .true.
     real(default) :: m = 0
    contains
    ⟨SF base: sf test spectrum data: TBP⟩
   end type sf_test_spectrum_data_t
```

Output.

⟨*SF base: sf test spectrum data: TBP*⟩≡
```
   procedure :: write => sf_test_spectrum_data_write
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_spectrum_data_write (data, unit, verbose)
      class(sf_test_spectrum_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "SF test spectrum data:"
      write (u, "(3x,A,A)") "model     = ", char (data%model%get_name ())
      write (u, "(3x,A)", advance="no") "incoming  = "
      call flavor_write (data%flv_in, u);  write (u, *)
      write (u, "(3x,A)", advance="no") "outgoing  = "
      call flavor_write (data%flv_out, u);  write (u, *)
      write (u, "(3x,A)", advance="no") "radiated  = "
      call flavor_write (data%flv_rad, u);  write (u, *)
      write (u, "(3x,A," // FMT_19 // ")")  "mass      = ", data%m
    end subroutine sf_test_spectrum_data_write
```

Initialization.

⟨*SF base: sf test spectrum data: TBP*⟩+≡
```
    procedure :: init => sf_test_spectrum_data_init
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_spectrum_data_init (data, model, pdg_in, with_radiation)
      class(sf_test_spectrum_data_t), intent(out) :: data
      type(model_t), intent(in), target :: model
      type(pdg_array_t), intent(in) :: pdg_in
      logical, intent(in) :: with_radiation
      data%model => model
      data%with_radiation = with_radiation
      if (pdg_array_get (pdg_in, 1) /= 25) then
         call msg_fatal ("Test structure function: input flavor must be 's'")
      end if
      call flavor_init (data%flv_in, 25, model)
      data%m = flavor_get_mass (data%flv_in)
      call flavor_init (data%flv_out, 25, model)
      if (with_radiation) then
         call flavor_init (data%flv_rad, 25, model)
      end if
    end subroutine sf_test_spectrum_data_init
```

Return the number of parameters: 2, since we have only collinear splitting here.

⟨*SF base: sf test spectrum data: TBP*⟩+≡
```
    procedure :: get_n_par => sf_test_spectrum_data_get_n_par
```

⟨*SF base: procedures*⟩+≡
```
    function sf_test_spectrum_data_get_n_par (data) result (n)
      class(sf_test_spectrum_data_t), intent(in) :: data
      integer :: n
      n = 2
    end function sf_test_spectrum_data_get_n_par
```

Return the outgoing particle PDG codes: 25

⟨*SF base: sf test spectrum data: TBP*⟩+≡
```
procedure :: get_pdg_out => sf_test_spectrum_data_get_pdg_out
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_spectrum_data_get_pdg_out (data, pdg_out)
  class(sf_test_spectrum_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  pdg_out(1) = 25
  pdg_out(2) = 25
end subroutine sf_test_spectrum_data_get_pdg_out
```

Allocate the matching interaction.

⟨*SF base: sf test spectrum data: TBP*⟩+≡
```
procedure :: allocate_sf_int => &
    sf_test_spectrum_data_allocate_sf_int
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_spectrum_data_allocate_sf_int (data, sf_int)
  class(sf_test_spectrum_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (sf_test_spectrum_t :: sf_int)
end subroutine sf_test_spectrum_data_allocate_sf_int
```

Interaction

⟨*SF base: test types*⟩+≡
```
type, extends (sf_int_t) :: sf_test_spectrum_t
  type(sf_test_spectrum_data_t), pointer :: data => null ()
 contains
 ⟨SF base: sf test spectrum: TBP⟩
end type sf_test_spectrum_t
```

⟨*SF base: sf test spectrum: TBP*⟩≡
```
procedure :: type_string => sf_test_spectrum_type_string
```

⟨*SF base: procedures*⟩+≡
```
function sf_test_spectrum_type_string (object) result (string)
  class(sf_test_spectrum_t), intent(in) :: object
  type(string_t) :: string
  string = "Test Spectrum"
end function sf_test_spectrum_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF base: sf test spectrum: TBP*⟩+≡
```
procedure :: write => sf_test_spectrum_write
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_spectrum_write (object, unit, testflag)
  class(sf_test_spectrum_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
```

1124

```
      u = output_unit (unit)
      if (associated (object%data)) then
         call object%data%write (u)
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "SF test spectrum data: [undefined]"
      end if
   end subroutine sf_test_spectrum_write
```

Initialize. We know that `data` will be of concrete type `sf_test_spectrum_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

⟨*SF base: sf test spectrum: TBP*⟩+≡
```
      procedure :: init => sf_test_spectrum_init
```

⟨*SF base: procedures*⟩+≡
```
   subroutine sf_test_spectrum_init (sf_int, data)
     class(sf_test_spectrum_t), intent(out) :: sf_int
     class(sf_data_t), intent(in), target :: data
     type(quantum_numbers_mask_t), dimension(6) :: mask
     type(helicity_t) :: hel0
     type(quantum_numbers_t), dimension(6) :: qn
     mask = new_quantum_numbers_mask (.false., .false., .false.)
     select type (data)
     type is (sf_test_spectrum_data_t)
        if (data%with_radiation) then
           call sf_int%base_init (mask(1:6), &
                [data%m**2, data%m**2], &
                [0._default, 0._default], &
                [data%m**2, data%m**2])
           sf_int%data => data
           call helicity_init (hel0, 0)
           call quantum_numbers_init (qn(1), data%flv_in,  hel0)
           call quantum_numbers_init (qn(2), data%flv_in,  hel0)
           call quantum_numbers_init (qn(3), data%flv_rad, hel0)
           call quantum_numbers_init (qn(4), data%flv_rad, hel0)
           call quantum_numbers_init (qn(5), data%flv_out, hel0)
           call quantum_numbers_init (qn(6), data%flv_out, hel0)
           call interaction_add_state (sf_int%interaction_t, qn(1:6))
           call sf_int%set_incoming ([1,2])
           call sf_int%set_radiated ([3,4])
           call sf_int%set_outgoing ([5,6])
        else
           call sf_int%base_init (mask(1:4), &
                [data%m**2, data%m**2], &
                [real(default) :: ], &
                [data%m**2, data%m**2])
           sf_int%data => data
           call helicity_init (hel0, 0)
           call quantum_numbers_init (qn(1), data%flv_in,  hel0)
           call quantum_numbers_init (qn(2), data%flv_in,  hel0)
```

```
            call quantum_numbers_init (qn(3), data%flv_out, hel0)
            call quantum_numbers_init (qn(4), data%flv_out, hel0)
            call interaction_add_state (sf_int%interaction_t, qn(1:4))
            call sf_int%set_incoming ([1,2])
            call sf_int%set_outgoing ([3,4])
         end if
         call interaction_freeze (sf_int%interaction_t)
      end select
      sf_int%status = SF_INITIAL
    end subroutine sf_test_spectrum_init
```

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ (as above) for both x parameters and consequently $f(r) = 4 r_1 r_2$.

⟨*SF base: sf test spectrum: TBP*⟩+≡
```
    procedure :: complete_kinematics => sf_test_spectrum_complete_kinematics
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_spectrum_complete_kinematics (sf_int, x, f, r, rb, map)
      class(sf_test_spectrum_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      real(default), dimension(2) :: xb1
      if (map) then
         x = r**2
         f = 4 * r(1) * r(2)
      else
         x = r
         f = 1
      end if
      if (sf_int%data%with_radiation) then
         xb1 = 1 - x
         call sf_int%split_momenta (x, xb1)
      else
         call sf_int%reduce_momenta (x)
      end if
      select case (sf_int%status)
      case (SF_FAILED_KINEMATICS);  f = 0
      end select
    end subroutine sf_test_spectrum_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF base: sf test spectrum: TBP*⟩+≡
```
    procedure :: inverse_kinematics => sf_test_spectrum_inverse_kinematics
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_spectrum_inverse_kinematics &
```

1126

```
      (sf_int, x, f, r, rb, map, set_momenta)
    class(sf_test_spectrum_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    real(default), dimension(:), intent(out) :: rb
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default), dimension(2) :: xb1
    logical :: set_mom
    set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
    if (map) then
       r = sqrt (x)
       f = 4 * r(1) * r(2)
    else
       r = x
       f = 1
    end if
    rb = 1 - r
    if (set_mom)  then
       if (sf_int%data%with_radiation) then
          xb1 = 1 - x
          call sf_int%split_momenta (x, xb1)
       else
          call sf_int%reduce_momenta (x)
       end if
       select case (sf_int%status)
       case (SF_FAILED_KINEMATICS);  f = 0
       end select
    end if
  end subroutine sf_test_spectrum_inverse_kinematics
```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

⟨*SF base: sf test spectrum: TBP*⟩+≡
```
    procedure :: apply => sf_test_spectrum_apply
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_spectrum_apply (sf_int, scale)
      class(sf_test_spectrum_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
      call interaction_set_matrix_element (sf_int%interaction_t, &
           cmplx (1._default, kind=default))
      sf_int%status = SF_EVALUATED
    end subroutine sf_test_spectrum_apply
```

10.3.13 Test implementation: generator spectrum

A generator for two beams, no radiation (for simplicity).

Configuration data

For simplicity, the spectrum contains two mirror images of the previous structure-function configuration: the incoming and all outgoing particles are test scalars.

We have two versions, one with radiated particles, one without.

⟨*SF base: test types*⟩+≡
```
type, extends (sf_data_t) :: sf_test_generator_data_t
  type(model_t), pointer :: model => null ()
  type(flavor_t) :: flv_in
  type(flavor_t) :: flv_out
  type(flavor_t) :: flv_rad
  real(default) :: m = 0
contains
⟨SF base: sf test generator data: TBP⟩
end type sf_test_generator_data_t
```

Output.

⟨*SF base: sf test generator data: TBP*⟩≡
```
procedure :: write => sf_test_generator_data_write
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_generator_data_write (data, unit, verbose)
  class(sf_test_generator_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)")  "SF test generator data:"
  write (u, "(3x,A,A)") "model     = ", char (data%model%get_name ())
  write (u, "(3x,A)", advance="no") "incoming  = "
  call flavor_write (data%flv_in, u);  write (u, *)
  write (u, "(3x,A)", advance="no") "outgoing  = "
  call flavor_write (data%flv_out, u);  write (u, *)
  write (u, "(3x,A," // FMT_19 // ")")  "mass      = ", data%m
end subroutine sf_test_generator_data_write
```

Initialization.

⟨*SF base: sf test generator data: TBP*⟩+≡
```
procedure :: init => sf_test_generator_data_init
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_test_generator_data_init (data, model, pdg_in)
  class(sf_test_generator_data_t), intent(out) :: data
  type(model_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  data%model => model
  if (pdg_array_get (pdg_in, 1) /= 25) then
     call msg_fatal ("Test generator: input flavor must be 's'")
  end if
  call flavor_init (data%flv_in, 25, model)
  data%m = flavor_get_mass (data%flv_in)
  call flavor_init (data%flv_out, 25, model)
end subroutine sf_test_generator_data_init
```

This structure function is a generator.

⟨SF base: sf test generator data: TBP⟩+≡
```
    procedure :: is_generator => sf_test_generator_data_is_generator
```

⟨SF base: procedures⟩+≡
```
    function sf_test_generator_data_is_generator (data) result (flag)
      class(sf_test_generator_data_t), intent(in) :: data
      logical :: flag
      flag = .true.
    end function sf_test_generator_data_is_generator
```

Return the number of parameters: 2, since we have only collinear splitting here.

⟨SF base: sf test generator data: TBP⟩+≡
```
    procedure :: get_n_par => sf_test_generator_data_get_n_par
```

⟨SF base: procedures⟩+≡
```
    function sf_test_generator_data_get_n_par (data) result (n)
      class(sf_test_generator_data_t), intent(in) :: data
      integer :: n
      n = 2
    end function sf_test_generator_data_get_n_par
```

Return the outgoing particle PDG codes: 25

⟨SF base: sf test generator data: TBP⟩+≡
```
    procedure :: get_pdg_out => sf_test_generator_data_get_pdg_out
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_test_generator_data_get_pdg_out (data, pdg_out)
      class(sf_test_generator_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      pdg_out(1) = 25
      pdg_out(2) = 25
    end subroutine sf_test_generator_data_get_pdg_out
```

Allocate the matching interaction.

⟨SF base: sf test generator data: TBP⟩+≡
```
    procedure :: allocate_sf_int => &
        sf_test_generator_data_allocate_sf_int
```

⟨SF base: procedures⟩+≡
```
    subroutine sf_test_generator_data_allocate_sf_int (data, sf_int)
      class(sf_test_generator_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
      allocate (sf_test_generator_t :: sf_int)
    end subroutine sf_test_generator_data_allocate_sf_int
```

Interaction

⟨SF base: test types⟩+≡
```
    type, extends (sf_int_t) :: sf_test_generator_t
      type(sf_test_generator_data_t), pointer :: data => null ()
     contains
```

⟨*SF base: sf test generator: TBP*⟩
```
      end type sf_test_generator_t
```

⟨*SF base: sf test generator: TBP*⟩≡
```
      procedure :: type_string => sf_test_generator_type_string
```

⟨*SF base: procedures*⟩+≡
```
      function sf_test_generator_type_string (object) result (string)
        class(sf_test_generator_t), intent(in) :: object
        type(string_t) :: string
        string = "Test Generator"
      end function sf_test_generator_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF base: sf test generator: TBP*⟩+≡
```
      procedure :: write => sf_test_generator_write
```

⟨*SF base: procedures*⟩+≡
```
      subroutine sf_test_generator_write (object, unit, testflag)
        class(sf_test_generator_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: testflag
        integer :: u
        u = output_unit (unit)
        if (associated (object%data)) then
           call object%data%write (u)
           call object%base_write (u, testflag)
        else
           write (u, "(1x,A)")  "SF test generator data: [undefined]"
        end if
      end subroutine sf_test_generator_write
```

Initialize. We know that `data` will be of concrete type `sf_test_generator_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass. No radiation.

⟨*SF base: sf test generator: TBP*⟩+≡
```
      procedure :: init => sf_test_generator_init
```

⟨*SF base: procedures*⟩+≡
```
      subroutine sf_test_generator_init (sf_int, data)
        class(sf_test_generator_t), intent(out) :: sf_int
        class(sf_data_t), intent(in), target :: data
        type(quantum_numbers_mask_t), dimension(4) :: mask
        type(helicity_t) :: hel0
        type(quantum_numbers_t), dimension(4) :: qn
        mask = new_quantum_numbers_mask (.false., .false., .false.)
        select type (data)
        type is (sf_test_generator_data_t)
           call sf_int%base_init (mask(1:4), &
                [data%m**2, data%m**2], &
                [real(default) :: ], &
                [data%m**2, data%m**2])
           sf_int%data => data
```

```
      call helicity_init (hel0, 0)
      call quantum_numbers_init (qn(1), data%flv_in,  hel0)
      call quantum_numbers_init (qn(2), data%flv_in,  hel0)
      call quantum_numbers_init (qn(3), data%flv_out, hel0)
      call quantum_numbers_init (qn(4), data%flv_out, hel0)
      call interaction_add_state (sf_int%interaction_t, qn(1:4))
      call sf_int%set_incoming ([1,2])
      call sf_int%set_outgoing ([3,4])
      call interaction_freeze (sf_int%interaction_t)
    end select
    sf_int%status = SF_INITIAL
  end subroutine sf_test_generator_init
```

This structure function is a generator.

⟨*SF base: sf test generator: TBP*⟩+≡
```
    procedure :: is_generator => sf_test_generator_is_generator
```

⟨*SF base: procedures*⟩+≡
```
    function sf_test_generator_is_generator (sf_int) result (flag)
      class(sf_test_generator_t), intent(in) :: sf_int
      logical :: flag
      flag = sf_int%data%is_generator ()
    end function sf_test_generator_is_generator
```

Generate free parameters. This mock generator always produces the nubmers 0.8 and 0.5.

⟨*SF base: sf test generator: TBP*⟩+≡
```
    procedure :: generate_free => sf_test_generator_generate_free
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_generator_generate_free (sf_int, r, rb,  x_free)
      class(sf_test_generator_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(inout) :: x_free
      r = [0.8, 0.5]
      rb= 1 - r
      x_free = x_free * product (r)
    end subroutine sf_test_generator_generate_free
```

Recover momentum fractions. Since the x values are free, we also set the `x_free` parameter.

⟨*SF base: sf test generator: TBP*⟩+≡
```
    procedure :: recover_x => sf_test_generator_recover_x
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_generator_recover_x (sf_int, x, x_free)
      class(sf_test_generator_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(inout), optional :: x_free
      call sf_int%base_recover_x (x)
      if (present (x_free))  x_free = x_free * product (x)
    end subroutine sf_test_generator_recover_x
```

Set kinematics. Since this is a generator, just transfer input to output.

⟨*SF base: sf test generator: TBP*⟩+≡
```
    procedure :: complete_kinematics => sf_test_generator_complete_kinematics
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_generator_complete_kinematics (sf_int, x, f, r, rb, map)
      class(sf_test_generator_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      x = r
      f = 1
      call sf_int%reduce_momenta (x)
    end subroutine sf_test_generator_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the
"input" r values and the Jacobian f. After this, we can set momenta by the
same formula as for normal kinematics.

⟨*SF base: sf test generator: TBP*⟩+≡
```
    procedure :: inverse_kinematics => sf_test_generator_inverse_kinematics
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_generator_inverse_kinematics &
        (sf_int, x, f, r, rb, map, set_momenta)
      class(sf_test_generator_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      r = x
      rb= 1 - x
      f = 1
      if (set_mom)  call sf_int%reduce_momenta (x)
    end subroutine sf_test_generator_inverse_kinematics
```

Apply the structure function. The matrix element becomes unity and the ap-
plication always succeeds.

⟨*SF base: sf test generator: TBP*⟩+≡
```
    procedure :: apply => sf_test_generator_apply
```

⟨*SF base: procedures*⟩+≡
```
    subroutine sf_test_generator_apply (sf_int, scale)
      class(sf_test_generator_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
      call interaction_set_matrix_element (sf_int%interaction_t, &
          cmplx (1._default, kind=default))
      sf_int%status = SF_EVALUATED
    end subroutine sf_test_generator_apply
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF base: execute tests*⟩≡

```
call test (sf_base_1, "sf_base_1", &
    "structure function configuration", &
    u, results)
```

⟨*SF base: tests*⟩+≡

```
subroutine sf_base_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_base_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
      &test structure function data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
      var_str ("Test.mdl"), os_data, model)
  pdg_in = 25

  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle code:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  write (u, "(2x,99(1x,I0))")  pdg1

  call model_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_base_1"

end subroutine sf_base_1
```

Test and probe structure function

Construct and display a structure function object based on the test structure function.

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_2, "sf_base_2", &
        "structure function instance", &
        u, results)
```

⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_base_2"
      write (u, "(A)")  "*   Purpose: initialize and fill &
          &test structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
          var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)
      pdg_in = 25

      call reset_interaction_counter ()

      allocate (sf_test_data_t :: data)
      select type (data)
      type is (sf_test_data_t)
         call data%init (model, pdg_in)
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
      call sf_int%set_beam_index ([1])
```

```
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize incoming momentum with E=500"
      write (u, "(A)")
      E = 500
      k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
      call vector4_write (k, u)
      call sf_int%seed_kinematics ([k])

      write (u, "(A)")
      write (u, "(A)")  "* Set kinematics for x=0"
      write (u, "(A)")

      allocate (r (data%get_n_par ()))
      allocate (rb(size (r)))
      allocate (x (size (r)))

      r = 0
      rb = 1 - r
      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Set kinematics for x=1"
      write (u, "(A)")

      r = 1
      rb = 1 - r
      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Set kinematics for x=0.5"
      write (u, "(A)")

      r = 0.5_default
      rb = 1 - r
      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
```

1135

```
    write (u, "(A)")  "* Set kinematics with mapping for r=0.8"
    write (u, "(A)")

    r = 0.8_default
    rb = 1 - r
    call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Recover x from momenta"
    write (u, "(A)")

    q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
    call sf_int%final ()
    deallocate (sf_int)

    call data%allocate_sf_int (sf_int)
    call sf_int%init (data)
    call sf_int%set_beam_index ([1])

    call sf_int%seed_kinematics ([k])
    call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
    call sf_int%recover_x (x)

    write (u, "(A,9(1x,F10.7))")  "x =", x

    write (u, "(A)")
    write (u, "(A)")  "* Compute inverse kinematics for x=0.64 and evaluate"
    write (u, "(A)")

    x = 0.64_default
    call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
    call sf_int%apply (scale=0._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_2"

  end subroutine sf_base_2
```

Collinear kinematics

Scan over the possibilities for mass assignment and on-shell projections, collinear case.

⟨*SF base: execute tests*⟩+≡
```
call test (sf_base_3, "sf_base_3", &
     "alternatives for collinear kinematics", &
     u, results)
```

⟨*SF base: tests*⟩+≡
```
subroutine sf_base_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(pdg_array_t) :: pdg_in
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_base_3"
  write (u, "(A)")  "*   Purpose: check various kinematical setups"
  write (u, "(A)")  "*            for collinear structure-function splitting."
  write (u, "(A)")  "            (two masses equal, one zero)"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)
  pdg_in = 25
  call flavor_init (flv, 25, model)

  call reset_interaction_counter ()

  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
```

1137

```
call sf_int%init (data)

call sf_int%write (u)

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"

E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set radiated mass to zero"

sf_int%mr2 = 0
sf_int%mo2 = sf_int%mi2

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")
```

```
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))") "x =", x
write (u, "(A,9(1x,F10.7))") "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set outgoing mass to zero"

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))") "x =", x
write (u, "(A,9(1x,F10.7))") "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))") "x =", x
write (u, "(A,9(1x,F10.7))") "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set incoming mass to zero"

k = vector4_moving (E, E, 3)
call sf_int%seed_kinematics ([k])
```

```
sf_int%mr2 = sf_int%mi2
sf_int%mo2 = sf_int%mi2
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set all masses to zero"

sf_int%mr2 = 0
sf_int%mo2 = 0
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")
```

```
        r = 0.5_default
        rb = 1 - r
        sf_int%on_shell_mode = KEEP_ENERGY
        call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
        call sf_int%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Recover x and r"
        write (u, "(A)")

        call sf_int%recover_x (x)
        call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
        write (u, "(A,9(1x,F10.7))")  "x =", x
        write (u, "(A,9(1x,F10.7))")  "r =", r

        write (u, "(A)")
        write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
        write (u, "(A)")

        r = 0.5_default
        rb = 1 - r
        sf_int%on_shell_mode = KEEP_MOMENTUM
        call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
        call sf_int%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Recover x and r"
        write (u, "(A)")

        call sf_int%recover_x (x)
        call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
        write (u, "(A,9(1x,F10.7))")  "x =", x
        write (u, "(A,9(1x,F10.7))")  "r =", r

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call sf_int%final ()
        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: sf_base_3"

    end subroutine sf_base_3
```

Non-collinear kinematics

Scan over the possibilities for mass assignment and on-shell projections, non-collinear case.

⟨*SF base: execute tests*⟩+≡
```
        call test (sf_base_4, "sf_base_4", &
```

```
               "alternatives for non-collinear kinematics", &
               u, results)
```

⟨*SF base: tests*⟩+≡

```
    subroutine sf_base_4 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      type(flavor_t) :: flv
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_base_4"
      write (u, "(A)")  "*   Purpose: check various kinematical setups"
      write (u, "(A)")  "*             for free structure-function splitting."
      write (u, "(A)")  "              (two masses equal, one zero)"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)
      pdg_in = 25

      call reset_interaction_counter ()

      allocate (sf_test_data_t :: data)
      select type (data)
      type is (sf_test_data_t)
         call data%init (model, pdg_in, collinear=.false.)
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)

      call sf_int%write (u)

      allocate (r (data%get_n_par ()))
      allocate (rb(size (r)))
      allocate (x (size (r)))

      write (u, "(A)")
```

1142

```
write (u, "(A)")  "* Initialize incoming momentum with E=500"

E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set radiated mass to zero"

sf_int%mr2 = 0
sf_int%mo2 = sf_int%mi2

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set outgoing mass to zero"

sf_int%mr2 = sf_int%mi2
```

```
sf_int%mo2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set incoming mass to zero"

k = vector4_moving (E, E, 3)
call sf_int%seed_kinematics ([k])

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = sf_int%mi2
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")
```

1144

```
r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set all masses to zero"

sf_int%mr2 = 0
sf_int%mo2 = 0
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Re-Initialize structure-function object with Q bounds"

call reset_interaction_counter ()

select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in, collinear=.false., &
        qbounds = [1._default, 100._default])
end select

call sf_int%init (data)
call sf_int%seed_kinematics ([k])
```

```
   write (u, "(A)")
   write (u, "(A)")   "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
   write (u, "(A)")

   r = [0.5_default, 0.5_default, 0.125_default]
   rb = 1 - r
   sf_int%on_shell_mode = KEEP_ENERGY
   call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
   call sf_int%write (u)

   write (u, "(A)")
   write (u, "(A)")   "* Recover x and r"
   write (u, "(A)")

   call sf_int%recover_x (x)
   call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
   write (u, "(A,9(1x,F10.7))")   "x =", x
   write (u, "(A,9(1x,F10.7))")   "r =", r

   write (u, "(A)")
   write (u, "(A)")   "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
   write (u, "(A)")

   r = [0.5_default, 0.5_default, 0.125_default]
   rb = 1 - r
   sf_int%on_shell_mode = KEEP_MOMENTUM
   call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
   call sf_int%write (u)

   write (u, "(A)")
   write (u, "(A)")   "* Recover x and r"
   write (u, "(A)")

   call sf_int%recover_x (x)
   call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
   write (u, "(A,9(1x,F10.7))")   "x =", x
   write (u, "(A,9(1x,F10.7))")   "r =", r

   write (u, "(A)")
   write (u, "(A)")   "* Cleanup"

   call sf_int%final ()
   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")   "* Test output end: sf_base_4"

end subroutine sf_base_4
```

Pair spectrum

Construct and display a structure function object for a pair spectrum (a structure function involving two particles simultaneously).

⟨*SF base: execute tests*⟩+≡
```
call test (sf_base_5, "sf_base_5", &
    "pair spectrum with radiation", &
    u, results)
```

⟨*SF base: tests*⟩+≡
```
subroutine sf_base_5 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(2) :: pdg_out
  integer, dimension(:), allocatable :: pdg1, pdg2
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t), dimension(2) :: k
  type(vector4_t), dimension(4) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_base_5"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &a pair spectrum object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)
  pdg_in = 25

  call reset_interaction_counter ()

  allocate (sf_test_spectrum_data_t :: data)
  select type (data)
  type is (sf_test_spectrum_data_t)
     call data%init (model, pdg_in, with_radiation=.true.)
  end select

  write (u, "(1x,A)")  "Outgoing particle codes:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  pdg2 = pdg_out(2)
  write (u, "(2x,99(1x,I0))")  pdg1, pdg2
```

```
write (u, "(A)")
write (u, "(A)")   "* Initialize spectrum object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")   "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")   "* Set kinematics for x=0.4,0.8"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.4_default, 0.8_default]
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")   "x =", x
write (u, "(A,9(1x,F10.7))")   "f =", f

write (u, "(A)")
write (u, "(A)")   "* Set kinematics with mapping for r=0.6,0.8"
write (u, "(A)")

r = [0.6_default, 0.8_default]
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")   "x =", x
write (u, "(A,9(1x,F10.7))")   "f =", f

write (u, "(A)")
write (u, "(A)")   "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
```

1148

```
        deallocate (sf_int)

        call reset_interaction_counter ()
        call data%allocate_sf_int (sf_int)
        call sf_int%init (data)

        call sf_int%seed_kinematics (k)
        call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
        call sf_int%recover_x (x)
        write (u, "(A,9(1x,F10.7))")  "x =", x

        write (u, "(A)")
        write (u, "(A)")  "* Compute inverse kinematics for x=0.36,0.64 &
             &and evaluate"
        write (u, "(A)")

        x = [0.36_default, 0.64_default]
        call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
        call sf_int%apply (scale=0._default)
        call sf_int%write (u)

        write (u, "(A)")
        write (u, "(A,9(1x,F10.7))")  "r =", r
        write (u, "(A,9(1x,F10.7))")  "f =", f

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call sf_int%final ()
        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: sf_base_5"

      end subroutine sf_base_5
```

Pair spectrum without radiation

Construct and display a structure function object for a pair spectrum (a structure function involving two particles simultaneously).

⟨*SF base: execute tests*⟩+≡
```
        call test (sf_base_6, "sf_base_6", &
             "pair spectrum without radiation", &
             u, results)
```

⟨*SF base: tests*⟩+≡
```
      subroutine sf_base_6 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_list_t) :: model_list
        type(model_t), pointer :: model
        type(pdg_array_t) :: pdg_in
```

```
type(flavor_t) :: flv
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(vector4_t), dimension(2) :: k
type(vector4_t), dimension(2) :: q
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: f

write (u, "(A)")  "* Test output: sf_base_6"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &a pair spectrum object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)
pdg_in = 25

call reset_interaction_counter ()

allocate (sf_test_spectrum_data_t :: data)
select type (data)
type is (sf_test_spectrum_data_t)
   call data%init (model, pdg_in, with_radiation=.false.)
end select

write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.4,0.8"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.4_default, 0.8_default]
```

```
      rb = 1 - r
      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Recover x from momenta"
      write (u, "(A)")

      q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
      call sf_int%final ()
      deallocate (sf_int)

      call reset_interaction_counter ()
      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)

      call sf_int%seed_kinematics (k)
      call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
      call sf_int%recover_x (x)
      write (u, "(A,9(1x,F10.7))")  "x =", x

      write (u, "(A)")
      write (u, "(A)")  "* Compute inverse kinematics for x=0.4,0.8 &
           &and evaluate"
      write (u, "(A)")

      x = [0.4_default, 0.8_default]
      call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
      call sf_int%apply (scale=0._default)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "r =", r
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_base_6"

  end subroutine sf_base_6
```

Direct access to structure function

Probe a structure function directly.

⟨*SF base: execute tests*⟩+≡
```
call test (sf_base_7, "sf_base_7", &
     "direct access", &
     u, results)
```

⟨*SF base: tests*⟩+≡
```
subroutine sf_base_7 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(pdg_array_t) :: pdg_in
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  real(default), dimension(:), allocatable :: value

  write (u, "(A)")  "* Test output: sf_base_7"
  write (u, "(A)")  "*   Purpose: check direct access method"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)
  call flavor_init (flv, 25, model)
  pdg_in = 25

  call reset_interaction_counter ()

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)

  write (u, "(A)")  "* Probe structure function: states"
  write (u, "(A)")

  write (u, "(A,I0)")  "n_states = ", sf_int%get_n_states ()
  write (u, "(A,I0)")  "n_in     = ", sf_int%get_n_in ()
  write (u, "(A,I0)")  "n_rad    = ", sf_int%get_n_rad ()
  write (u, "(A,I0)")  "n_out    = ", sf_int%get_n_out ()
```

1152

```
write (u, "(A)")
write (u, "(A)", advance="no")  "state(1)  = "
call quantum_numbers_write (sf_int%get_state (1), u)
write (u, *)

allocate (value (sf_int%get_n_states ()))
call sf_int%compute_values (value, &
     E=[500._default], x=[0.5_default], xb=[0.5_default], scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no")  "value (E=500, x=0.5) ="
write (u, "(9(1x," // FMT_19 // "))")  value

call sf_int%compute_values (value, &
     x=[0.1_default], xb=[0.9_default], scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no")  "value (E=500, x=0.1) ="
write (u, "(9(1x," // FMT_19 // "))")  value


write (u, "(A)")
write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

deallocate (value)
call sf_int%final ()
deallocate (sf_int)
deallocate (data)

allocate (sf_test_spectrum_data_t :: data)
select type (data)
type is (sf_test_spectrum_data_t)
   call data%init (model, pdg_in, with_radiation=.false.)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Probe spectrum: states"
write (u, "(A)")

write (u, "(A,I0)")  "n_states = ", sf_int%get_n_states ()
write (u, "(A,I0)")  "n_in     = ", sf_int%get_n_in ()
write (u, "(A,I0)")  "n_rad    = ", sf_int%get_n_rad ()
write (u, "(A,I0)")  "n_out    = ", sf_int%get_n_out ()
write (u, "(A)")
write (u, "(A)", advance="no")  "state(1)  = "
call quantum_numbers_write (sf_int%get_state (1), u)
write (u, *)

allocate (value (sf_int%get_n_states ()))
call sf_int%compute_value (1, value(1), &
     E = [500._default, 500._default], &
```

```
              x = [0.5_default, 0.6_default], &
              xb= [0.5_default, 0.4_default], &
              scale = 0._default)

      write (u, "(A)")
      write (u, "(A)", advance="no")  "value (E=500,500, x=0.5,0.6) ="
      write (u, "(9(1x," // FMT_19 // "))")  value

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_base_7"

    end subroutine sf_base_7
```

Structure function chain configuration

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_8, "sf_base_8", &
        "structure function chain configuration", &
        u, results)
```

⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_8 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      type(beam_data_t), target :: beam_data
      class(sf_data_t), allocatable, target :: data_strfun
      class(sf_data_t), allocatable, target :: data_spectrum
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_chain_t) :: sf_chain

      write (u, "(A)")  "* Test output: sf_base_8"
      write (u, "(A)")  "*   Purpose: set up a structure-function chain"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
          var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)
```

1154

```
pdg_in = 25

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
     1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
   call data_strfun%init (model, pdg_in)
end select

allocate (sf_test_spectrum_data_t :: data_spectrum)
select type (data_spectrum)
type is (sf_test_spectrum_data_t)
   call data_spectrum%init (model, pdg_in, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)
call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
```

1155

```
      call sf_chain%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_base_8"

    end subroutine sf_base_8
```

Structure function instance configuration

We create a structure-function chain instance which implements a configured
structure-function chain. We link the momentum entries in the interactions and
compute kinematics.

   We do not actually connect the interactions and create evaluators. We skip
this step and manually advance the status of the chain instead.

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_9, "sf_base_9", &
         "structure function chain instance", &
         u, results)
```
⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_9 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      type(beam_data_t), target :: beam_data
      class(sf_data_t), allocatable, target :: data_strfun
      class(sf_data_t), allocatable, target :: data_spectrum
      type(sf_config_t), dimension(:), allocatable, target :: sf_config
      type(sf_chain_t), target :: sf_chain
      type(sf_chain_instance_t), target :: sf_chain_instance
      type(sf_channel_t), dimension(2) :: sf_channel
      type(vector4_t), dimension(2) :: p
      integer :: j

      write (u, "(A)")  "* Test output: sf_base_9"
      write (u, "(A)")  "*   Purpose: set up a structure-function chain &
          &and create an instance"
      write (u, "(A)")  "*            compute kinematics"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
```

1156

```
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
     1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
   call data_strfun%init (model, pdg_in)
end select

allocate (sf_test_spectrum_data_t :: data_spectrum)
select type (data_spectrum)
type is (sf_test_spectrum_data_t)
   call data_spectrum%init (model, pdg_in, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [real(default) ::])

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Outgoing momenta:"

do j = 1, 2
   write (u, "(A)")
   call vector4_write (p(j), u)
end do

call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
```

```
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [0.8_default])

call write_separator_double (u)
call sf_chain%write (u)
call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Outgoing momenta:"

do j = 1, 2
   write (u, "(A)")
   call vector4_write (p(j), u)
end do

call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
```

```
      call sf_chain_instance%compute_kinematics &
           (1, [0.5_default, 0.6_default, 0.8_default])

      call write_separator_double (u)
      call sf_chain%write (u)
      call write_separator_double (u)
      call sf_chain_instance%write (u)
      call write_separator_double (u)

      call sf_chain_instance%get_out_momenta (p)

      write (u, "(A)")
      write (u, "(A)")  "* Outgoing momenta:"

      do j = 1, 2
         write (u, "(A)")
         call vector4_write (p(j), u)
      end do

      call sf_chain_instance%final ()
      call sf_chain%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_base_9"

    end subroutine sf_base_9
```

Structure function chain mappings

Set up a structure function chain instance with a pair of single-particle structure functions. We test different global mappings for this setup.

Again, we skip evaluators.

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_10, "sf_base_10", &
         "structure function chain mapping", &
         u, results)
```

⟨*SF base: tests*⟩+≡
```
  subroutine sf_base_10 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    type(beam_data_t), target :: beam_data
    class(sf_data_t), allocatable, target :: data_strfun
```

```
type(sf_config_t), dimension(:), allocatable, target :: sf_config
type(sf_chain_t), target :: sf_chain
type(sf_chain_instance_t), target :: sf_chain_instance
type(sf_channel_t), dimension(2) :: sf_channel
real(default), dimension(2) :: x_saved

write (u, "(A)")  "* Test output: sf_base_10"
write (u, "(A)")  "*   Purpose: set up a structure-function chain"
write (u, "(A)")  "*            and check mappings"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
     1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
   call data_strfun%init (model, pdg_in)
end select

write (u, "(A)")  "* Set up chain with structure function pair &
     &and standard mapping"
write (u, "(A)")

allocate (sf_config (2))
call sf_config(1)%init ([1], data_strfun)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(1)%init (2)
call sf_channel(1)%set_s_mapping ([1,2])
call sf_chain_instance%set_channel (1, sf_channel(1))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Invert the kinematics calculation"
    write (u, "(A)")

    x_saved = sf_chain_instance%x

    call sf_chain_instance%init (sf_chain, n_channel = 1)

    call sf_channel(2)%init (2)
    call sf_channel(2)%set_s_mapping ([1, 2])
    call sf_chain_instance%set_channel (1, sf_channel(2))

    call sf_chain_instance%link_interactions ()
    sf_chain_instance%status = SF_DONE_CONNECTIONS
    call sf_chain_instance%inverse_kinematics (x_saved)

    call write_separator_double (u)
    call sf_chain_instance%write (u)
    call write_separator_double (u)


    call sf_chain_instance%final ()
    call sf_chain%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_10"

  end subroutine sf_base_10
```

Structure function chain evaluation

Here, we test the complete workflow for structure-function chains. First, we create the template chain, then initialize an instance. We set up links, mask, and evaluators. Finally, we set kinematics and evaluate the matrix elements and their products.

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_11, "sf_base_11", &
         "structure function chain evaluation", &
         u, results)
```

⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_11 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
```

1161

```
type(flavor_t) :: flv
type(pdg_array_t) :: pdg_in
type(beam_data_t), target :: beam_data
class(sf_data_t), allocatable, target :: data_strfun
class(sf_data_t), allocatable, target :: data_spectrum
type(sf_config_t), dimension(:), allocatable, target :: sf_config
type(sf_chain_t), target :: sf_chain
type(sf_chain_instance_t), target :: sf_chain_instance
type(sf_channel_t), dimension(2) :: sf_channel
type(particle_set_t) :: pset
type(interaction_t), pointer :: int
logical :: ok

write (u, "(A)")  "* Test output: sf_base_11"
write (u, "(A)")  "*   Purpose: set up a structure-function chain"
write (u, "(A)")  "*           create an instance and evaluate"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
     1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
   call data_strfun%init (model, pdg_in)
end select

allocate (sf_test_spectrum_data_t :: data_spectrum)
select type (data_spectrum)
type is (sf_test_spectrum_data_t)
   call data_spectrum%init (model, pdg_in, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()
```

```
call sf_chain_instance%compute_kinematics (1, [real(default) ::])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
     [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_fill_interaction (pset, int, 2)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call particle_set_final (pset)
call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)
```

```
call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics (1, [0.8_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
     [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_fill_interaction (pset, int, 2)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

call particle_set_final (pset)
call sf_chain_instance%final ()
call sf_chain%final ()
```

```
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")   "* Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics &
     (1, [0.5_default, 0.6_default, 0.8_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

int => sf_chain_instance%get_out_int_ptr ()
call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
     [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")   "* Particle content:"
write (u, "(A)")

call write_separator (u)
call particle_set_write (pset, u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")   "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
```

```
      call particle_set_fill_interaction (pset, int, 2)

      call sf_chain_instance%recover_kinematics (1)
      call sf_chain_instance%evaluate (scale=0._default)

      call write_separator_double (u)
      call sf_chain_instance%write (u)
      call write_separator_double (u)

      call particle_set_final (pset)
      call sf_chain_instance%final ()
      call sf_chain%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_base_11"

    end subroutine sf_base_11
```

Multichannel case

We set up a structure-function chain as before, but with three different parameterizations. The first instance is without mappings, the second one with single-particle mappings, and the third one with two-particle mappings.

⟨*SF base: execute tests*⟩+≡
```
      call test (sf_base_12, "sf_base_12", &
          "multi-channel structure function chain", &
          u, results)
```

⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_12 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      type(beam_data_t), target :: beam_data
      class(sf_data_t), allocatable, target :: data
      type(sf_config_t), dimension(:), allocatable, target :: sf_config
      type(sf_chain_t), target :: sf_chain
      type(sf_chain_instance_t), target :: sf_chain_instance
      real(default), dimension(2) :: x_saved
      real(default), dimension(2,3) :: p_saved
      type(sf_channel_t), dimension(:), allocatable :: sf_channel

      write (u, "(A)")  "* Test output: sf_base_12"
      write (u, "(A)")  "*   Purpose: set up and evaluate a multi-channel &
```

```
          &structure-function chain"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
     1000._default, [flv, flv])

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in)
end select

write (u, "(A)")  "* Set up chain with structure function pair &
     &and three different mappings"
write (u, "(A)")

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 3)

call allocate_sf_channels (sf_channel, n_channel = 3, n_strfun = 2)

! channel 1: no mapping
call sf_chain_instance%set_channel (1, sf_channel(1))

! channel 2: single-particle mappings
call sf_channel(2)%activate_mapping ([1,2])
! call sf_chain_instance%activate_mapping (2, [1,2])
call sf_chain_instance%set_channel (2, sf_channel(2))

! channel 3: two-particle mapping
call sf_channel(3)%set_s_mapping ([1,2])
! call sf_chain_instance%set_s_mapping (3, [1, 2])
call sf_chain_instance%set_channel (3, sf_channel(3))

call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()
```

```
write (u, "(A)")  "* Compute kinematics in channel 1 and evaluate"
write (u, "(A)")

call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Invert the kinematics calculation"
write (u, "(A)")

x_saved = sf_chain_instance%x

call sf_chain_instance%inverse_kinematics (x_saved)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Compute kinematics in channel 2 and evaluate"
write (u, "(A)")

p_saved = sf_chain_instance%p

call sf_chain_instance%compute_kinematics (2, p_saved(:,2))
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Compute kinematics in channel 3 and evaluate"
write (u, "(A)")

call sf_chain_instance%compute_kinematics (3, p_saved(:,3))
call sf_chain_instance%evaluate (scale=0._default)

call write_separator_double (u)
call sf_chain_instance%write (u)
call write_separator_double (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_chain_instance%final ()
call sf_chain%final ()

call model_list%final ()
```

1168

```
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_12"

  end subroutine sf_base_12
```

Generated spectrum

Construct and evaluate a structure function object for a pair spectrum which is
evaluated as a beam-event generator.

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_13, "sf_base_13", &
         "pair spectrum generator", &
         u, results)
```

⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_13 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t), dimension(2) :: k
      type(vector4_t), dimension(2) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f, x_free

      write (u, "(A)")  "* Test output: sf_base_13"
      write (u, "(A)")  "*   Purpose: initialize and fill &
           &a pair generator object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)
      pdg_in = 25

      call reset_interaction_counter ()

      allocate (sf_test_generator_data_t :: data)
      select type (data)
      type is (sf_test_generator_data_t)
         call data%init (model, pdg_in)
```

1169

```
      end select

      write (u, "(A)")  "* Initialize generator object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)

      allocate (r (data%get_n_par ()))
      allocate (rb(size (r)))
      allocate (x (size (r)))

      write (u, "(A)")  "* Generate free r values"
      write (u, "(A)")

      x_free = 1
      call sf_int%generate_free (r, rb, x_free)

      write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

      E = 500
      k(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
      k(2) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
      call sf_int%seed_kinematics (k)

      write (u, "(A)")
      write (u, "(A)")  "* Complete kinematics"
      write (u, "(A)")

      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f
      write (u, "(A,9(1x,F10.7))")  "xf=", x_free

      write (u, "(A)")
      write (u, "(A)")  "* Recover x from momenta"
      write (u, "(A)")

      q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
      call sf_int%final ()
      deallocate (sf_int)

      call reset_interaction_counter ()
      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)

      call sf_int%seed_kinematics (k)
      call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
      x_free = 1
      call sf_int%recover_x (x, x_free)
      write (u, "(A,9(1x,F10.7))")  "x =", x
```

```
      write (u, "(A,9(1x,F10.7))")  "xf=", x_free

      write (u, "(A)")
      write (u, "(A)")  "* Compute inverse kinematics &
           &and evaluate"
      write (u, "(A)")

      call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
      call sf_int%apply (scale=0._default)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "r =", r
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_base_13"

    end subroutine sf_base_13
```

Structure function chain evaluation

Here, we test the complete workflow for a structure-function chain with generator. First, we create the template chain, then initialize an instance. We set up links, mask, and evaluators. Finally, we set kinematics and evaluate the matrix elements and their products.

⟨*SF base: execute tests*⟩+≡
```
    call test (sf_base_14, "sf_base_14", &
        "structure function generator evaluation", &
        u, results)
```

⟨*SF base: tests*⟩+≡
```
    subroutine sf_base_14 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      type(beam_data_t), target :: beam_data
      class(sf_data_t), allocatable, target :: data_strfun
      class(sf_data_t), allocatable, target :: data_generator
      type(sf_config_t), dimension(:), allocatable, target :: sf_config
      real(default), dimension(:), allocatable :: p_in
      type(sf_chain_t), target :: sf_chain
      type(sf_chain_instance_t), target :: sf_chain_instance
```

```
write (u, "(A)")  "* Test output: sf_base_14"
write (u, "(A)")  "*   Purpose: set up a structure-function chain"
write (u, "(A)")  "*            create an instance and evaluate"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data_init_sqrts (beam_data, &
     1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
   call data_strfun%init (model, pdg_in)
end select

allocate (sf_test_generator_data_t :: data_generator)
select type (data_generator)
type is (sf_test_generator_data_t)
   call data_generator%init (model, pdg_in)
end select

write (u, "(A)")  "* Set up chain with generator and structure function"
write (u, "(A)")

allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_generator)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

write (u, "(A)")  "* Inject integration parameter"
write (u, "(A)")

allocate (p_in (sf_chain%get_n_bound ()), source = 0.9_default)
write (u, "(A,9(1x,F10.7))")  "p_in =", p_in

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
```

```
         write (u, "(A)")

         call sf_chain_instance%compute_kinematics (1, p_in)
         call sf_chain_instance%evaluate (scale=0._default)

         call sf_chain_instance%write (u)

         write (u, "(A)")
         write (u, "(A)")   "* Extract integration parameter"
         write (u, "(A)")

         call sf_chain_instance%get_mcpar (1, p_in)
         write (u, "(A,9(1x,F10.7))")   "p_in =", p_in

         call sf_chain_instance%final ()
         call sf_chain%final ()

         write (u, "(A)")
         write (u, "(A)")   "* Cleanup"

         call model_list%final ()
         call syntax_model_file_final ()

         write (u, "(A)")
         write (u, "(A)")   "* Test output end: sf_base_14"

    end subroutine sf_base_14
```

10.4 Photon radiation: ISR

⟨sf_isr.f90⟩≡
  ⟨File header⟩

```
  module sf_isr
```

  ⟨Use kinds⟩
  ⟨Use strings⟩
```
    use constants, only: pi !NODEP!
```
  ⟨Use file utils⟩
```
    use limits, only: FMT_12, FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use sm_physics, only: Li2 !NODEP!
    use unit_tests
    use os_interface
    use pdg_arrays
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
```

```
    use interactions
    use sf_mappings
    use sf_aux
    use sf_base
```

⟨*Standard module head*⟩

⟨*SF isr: public*⟩

⟨*SF isr: parameters*⟩

⟨*SF isr: types*⟩

```
contains
```

⟨*SF isr: procedures*⟩

⟨*SF isr: tests*⟩

```
end module sf_isr
```

10.4.1   Physics

The ISR structure function is in the most crude approximation (LLA without α corrections, i.e. ϵ^0)

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \qquad \text{with} \qquad \epsilon = \frac{\alpha}{\pi} q_e^2 \ln \frac{s}{m^2}, \qquad (10.27)$$

where m is the mass of the incoming (and outgoing) particle, which is initially assumed on-shell.

Here, the form of ϵ results from the kinematical bounds for the momentum squared of the outgoing particle, which in the limit $m^2 \ll s$ are given by

$$t_0 = -2\bar{x}E(E+p) + m^2 \approx -\bar{x}s, \qquad (10.28)$$
$$t_1 = -2\bar{x}E(E-p) + m^2 \approx xm^2, \qquad (10.29)$$

so the integration over the propagator $1/(t-m^2)$ yields

$$\ln \frac{t_0 - m^2}{t_1 - m^2} = \ln \frac{s}{m^2}. \qquad (10.30)$$

In $f_0(x)$, there is an integrable singularity at $x = 1$ which does not spoil the integration, but would lead to an unbounded $f_{\max}$. Therefore, we map this singularity like

$$x = 1 - (1-x')^{1/\epsilon} \qquad (10.31)$$

such that

$$\int dx\, f_0(x) = \int dx' \qquad (10.32)$$

The structure function has three parameters: α, m_{in} of the incoming particle and s, the hard scale. Internally, we store the exponent ϵ which is the relevant parameter. (In conventional notation, $\epsilon = \beta/2$.) As defaults, we take the actual

1174

values of α (which is probably $\alpha(s)$), the actual mass m_{in} and the squared total c.m. energy s.

Including ϵ, ϵ^2, and ϵ^3 corrections, the successive approximation of the ISR structure function read

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \tag{10.33}$$

$$f_1(x) = g_1(\epsilon) f_0(x) - \frac{\epsilon}{2}(1+x) \tag{10.34}$$

$$f_2(x) = g_2(\epsilon) f_0(x) - \frac{\epsilon}{2}(1+x)$$
$$- \frac{\epsilon^2}{8}\left(\frac{1+3x^2}{1-x}\ln x + 4(1+x)\ln(1-x) + 5 + x\right) \tag{10.35}$$

$$f_3(x) = g_3(\epsilon) f_0(x) - \frac{\epsilon}{2}(1+x)$$
$$- \frac{\epsilon^2}{8}\left(\frac{1+3x^2}{1-x}\ln x + 4(1+x)\ln(1-x) + 5 + x\right)$$
$$- \frac{\epsilon^3}{48}\Bigg((1+x)\left[6\operatorname{Li}_2(x) + 12\ln^2(1-x) - 3\pi^2\right] + 6(x+5)\ln(1-x)$$
$$+ \frac{1}{1-x}\left[\frac{3}{2}(1+8x+3x^2)\ln x + 12(1+x^2)\ln x \ln(1-x)\right.$$
$$\left.- \frac{1}{2}(1+7x^2)\ln^2 x + \frac{1}{4}(39-24x-15x^2)\right]\Bigg) \tag{10.36}$$

where the successive approximations to the prefactor of the leading singularity

$$g(\epsilon) = \frac{\exp\left(\epsilon(-\gamma_E + \frac{3}{4})\right)}{\Gamma(1+\epsilon)}, \tag{10.37}$$

are given by

$$g_0(\epsilon) = 1 \tag{10.38}$$

$$g_1(\epsilon) = 1 + \frac{3}{4}\epsilon \tag{10.39}$$

$$g_2(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 \tag{10.40}$$

$$g_3(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 + \frac{27 - 24\pi^2 + 128\zeta(3)}{384}\epsilon^3, \tag{10.41}$$

where, numerically

$$\zeta(3) = 1.2020569031595942853997381 6151\ldots \tag{10.42}$$

Although one could calculate the function $g(\epsilon)$ exactly, truncating its Taylor expansion ensures the exact normalization of the truncated structure function at each given order:

$$\int_0^1 dx\, f_i(x) = 1 \qquad \text{for all } i. \tag{10.43}$$

Effectively, the $O(\epsilon)$ correction reduces the low-x tail of the structure function by 50% while increasing the coefficient of the singularity by $O(\epsilon)$. Relative

to this, the $O(\epsilon^2)$ correction slightly enhances $x > \frac{1}{2}$ compared to $x < \frac{1}{2}$. At $x = 0$, $f_2(x)$ introduces a logarithmic singularity which should be cut off at $x_0 = O(e^{-1/\epsilon})$: for lower x the perturbative series breaks down. The f_3 correction is slightly positive for low x values and negative near $x = 1$, where the Li$_2$ piece slightly softens the singularity at $x = 1$.

Instead of the definition for ϵ given above, it is customary to include a universal nonlogarithmic piece:

$$\epsilon = \frac{\alpha}{\pi} q_e^2 \left(\ln \frac{s}{m^2} - 1 \right) \tag{10.44}$$

10.4.2 Implementation

In the concrete implementation, the zeroth order mapping (10.31) is implemented, and the Jacobian is equal to $f_i(x)/f_0(x)$. This can be written as

$$\frac{f_0(x)}{f_0(x)} = 1 \tag{10.45}$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon - \frac{1 - x^2}{2(1 - x')} \tag{10.46}$$

$$\frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 - \frac{1 - x^2}{2(1 - x')}$$
$$- \frac{(1 + 3x^2)\ln x + (1 - x)\left(4(1 + x)\ln(1 - x) + 5 + x\right)}{8(1 - x')}\epsilon \tag{10.47}$$

For $x = 1$ (i.e., numerically indistinguishable from 1), this reduces to

$$\frac{f_0(x)}{f_0(x)} = 1 \tag{10.48}$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon \tag{10.49}$$

$$\frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 \tag{10.50}$$

The last line in (10.47) is zero for

$$x_{\min} = 0.0071405332973459283954987972019 \tag{10.51}$$

(Mathematica result), independent of ϵ. For x values less than this we ignore this correction because of the logarithmic singularity which should in principle be resummed.

10.4.3 The ISR data block

$\langle SF\ isr:\ public\rangle\equiv$
```
    public :: isr_data_t
```

$\langle SF\ isr:\ types\rangle\equiv$
```
    type, extends (sf_data_t) :: isr_data_t
      private
      type(model_t), pointer :: model => null ()
      type(flavor_t), dimension(:), allocatable :: flv_in
```

```
        real(default) :: alpha = 0
        real(default) :: q_max = 0
        real(default) :: real_mass = 0
        real(default) :: mass = 0
        real(default) :: eps = 0
        real(default) :: log = 0
        logical :: recoil = .false.
        integer :: order = 3
        integer :: error = NONE
      contains
      ⟨SF isr: isr data: TBP⟩
      end type isr_data_t
```

Error codes

⟨*SF isr: parameters*⟩≡
```
    integer, parameter :: NONE = 0
    integer, parameter :: ZERO_MASS = 1
    integer, parameter :: Q_MAX_TOO_SMALL = 2
    integer, parameter :: EPS_TOO_LARGE = 3
    integer, parameter :: INVALID_ORDER = 4
    integer, parameter :: CHARGE_MIX = 5
    integer, parameter :: CHARGE_ZERO = 6
    integer, parameter :: MASS_MIX = 7
```
Generate flavor-dependent ISR data:

⟨*SF isr: isr data: TBP*⟩≡
```
    procedure :: init => isr_data_init
```

⟨*SF isr: procedures*⟩≡
```
    subroutine isr_data_init &
        (data, model, pdg_in, alpha, q_max, mass, order, recoil)
      class(isr_data_t), intent(out) :: data
      type(model_t), intent(in), target :: model
      type(pdg_array_t), intent(in) :: pdg_in
      real(default), intent(in) :: alpha
      real(default), intent(in) :: q_max
      real(default), intent(in), optional :: mass
      integer, intent(in), optional :: order
      logical, intent(in), optional :: recoil
      integer :: i, n_flv
      real(default) :: charge
      data%model => model
      n_flv = pdg_array_get_length (pdg_in)
      allocate (data%flv_in (n_flv))
      do i = 1, n_flv
         call flavor_init (data%flv_in(i), pdg_array_get (pdg_in, i), model)
      end do
      data%alpha = alpha
      data%q_max = q_max
      if (present (order)) then
         call data%set_order (order)
      end if
      if (present (recoil)) then
         data%recoil = recoil
```

```
         end if
         data%real_mass = flavor_get_mass (data%flv_in(1))
         if (present (mass)) then
            if (mass > 0) then
               data%mass = mass
            else
               data%mass = data%real_mass
               if (any (flavor_get_mass (data%flv_in) /= data%mass)) then
                  data%error = MASS_MIX;  return
               end if
            end if
         else
            data%mass = data%real_mass
            if (any (flavor_get_mass (data%flv_in) /= data%mass)) then
               data%error = MASS_MIX;  return
            end if
         end if
         if (data%mass == 0) then
            data%error = ZERO_MASS;  return
         else if (data%mass >= data%q_max) then
            data%error = Q_MAX_TOO_SMALL;  return
         end if
         data%log = log (1 + (data%q_max / data%mass)**2)
         charge = flavor_get_charge (data%flv_in(1))
         if (any (abs (flavor_get_charge (data%flv_in)) /= abs (charge))) then
            data%error = CHARGE_MIX;  return
         else if (charge == 0) then
            data%error = CHARGE_ZERO;  return
         end if
         data%eps = data%alpha / pi * charge ** 2 &
              * (2 * log (data%q_max / data%mass) - 1)
         if (data%eps > 1) then
            data%error = EPS_TOO_LARGE;  return
         end if
      end subroutine isr_data_init
```

Explicitly set ISR order

⟨*SF isr: isr data: TBP*⟩+≡

```
      procedure :: set_order => isr_data_set_order
```

⟨*SF isr: procedures*⟩+≡

```
      elemental subroutine isr_data_set_order (data, order)
        class(isr_data_t), intent(inout) :: data
        integer, intent(in) :: order
        if (order < 0 .or. order > 3) then
           data%error = INVALID_ORDER
        else
           data%order = order
        end if
      end subroutine isr_data_set_order
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*SF isr: isr data: TBP*⟩+≡

```
        procedure :: check => isr_data_check
```

⟨*SF isr: procedures*⟩+≡
```
    subroutine isr_data_check (data)
      class(isr_data_t), intent(in) :: data
      select case (data%error)
      case (ZERO_MASS)
         call msg_fatal ("ISR: Particle mass is zero")
      case (Q_MAX_TOO_SMALL)
         call msg_fatal ("ISR: Particle mass exceeds Qmax")
      case (EPS_TOO_LARGE)
         call msg_fatal ("ISR: Expansion parameter too large, " // &
              "perturbative expansion breaks down")
      case (INVALID_ORDER)
         call msg_error ("ISR: LLA order invalid (valid values are 0,1,2,3)")
      case (MASS_MIX)
         call msg_fatal ("ISR: Incoming particle masses must be uniform")
      case (CHARGE_MIX)
         call msg_fatal ("ISR: Incoming particle charges must be uniform")
      case (CHARGE_ZERO)
         call msg_fatal ("ISR: Incoming particle must be charged")
      end select
    end subroutine isr_data_check
```

Output

⟨*SF isr: isr data: TBP*⟩+≡
```
    procedure :: write => isr_data_write
```

⟨*SF isr: procedures*⟩+≡
```
    subroutine isr_data_write (data, unit, verbose)
      class(isr_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)") "ISR data:"
      if (allocated (data%flv_in)) then
         write (u, "(3x,A)", advance="no") "  flavor =  "
         do i = 1, size (data%flv_in)
            if (i > 1)  write (u, "(',',1x)", advance="no")
            call flavor_write (data%flv_in(i), u)
         end do
         write (u, *)
         write (u, "(3x,A," // FMT_19 // ")") "  alpha  = ", data%alpha
         write (u, "(3x,A," // FMT_19 // ")") "  q_max  = ", data%q_max
         write (u, "(3x,A," // FMT_19 // ")") "  mass   = ", data%mass
         write (u, "(3x,A," // FMT_19 // ")") "  eps    = ", data%eps
         write (u, "(3x,A," // FMT_19 // ")") "  log    = ", data%log
         write (u, "(3x,A,I2)")        "  order  = ", data%order
         write (u, "(3x,A,L2)")        "  recoil = ", data%recoil
      else
         write (u, "(3x,A)") "[undefined]"
      end if
    end subroutine isr_data_write
```

For ISR, there is the option to generate transverse momentum is generated. Hence, there can be up to three parameters, x, and two angles.

⟨*SF isr: isr data: TBP*⟩+≡
```
    procedure :: get_n_par => isr_data_get_n_par
```

⟨*SF isr: procedures*⟩+≡
```
    function isr_data_get_n_par (data) result (n)
      class(isr_data_t), intent(in) :: data
      integer :: n
      if (data%recoil) then
         n = 3
      else
         n = 1
      end if
    end function isr_data_get_n_par
```

Return the outgoing particles PDG codes. For ISR, these are identical to the incoming particles.

⟨*SF isr: isr data: TBP*⟩+≡
```
    procedure :: get_pdg_out => isr_data_get_pdg_out
```

⟨*SF isr: procedures*⟩+≡
```
    subroutine isr_data_get_pdg_out (data, pdg_out)
      class(isr_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      pdg_out(1) = flavor_get_pdg (data%flv_in)
    end subroutine isr_data_get_pdg_out
```

Return the `eps` value. We need it for an appropriate mapping of structure-function parameters.

⟨*SF isr: isr data: TBP*⟩+≡
```
    procedure :: get_eps => isr_data_get_eps
```

⟨*SF isr: procedures*⟩+≡
```
    function isr_data_get_eps (data) result (eps)
      class(isr_data_t), intent(in) :: data
      real(default) :: eps
      eps = data%eps
    end function isr_data_get_eps
```

Allocate the interaction record.

⟨*SF isr: isr data: TBP*⟩+≡
```
    procedure :: allocate_sf_int => isr_data_allocate_sf_int
```

⟨*SF isr: procedures*⟩+≡
```
    subroutine isr_data_allocate_sf_int (data, sf_int)
      class(isr_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
      allocate (isr_t :: sf_int)
    end subroutine isr_data_allocate_sf_int
```

10.4.4 The ISR object

The `isr_t` data type is a $1 \to 2$ interaction, i.e., we allow for single-photon emission only (but use the multi-photon resummed radiator function). The particles are ordered as (incoming, photon, outgoing).

There is no need to handle several flavors (and data blocks) in parallel, since ISR is always applied immediately after beam collision. (ISR for partons is accounted for by the PDFs themselves.) Polarization is carried through, i.e., we retain the polarization of the incoming particle and treat the emitted photon as unpolarized. Color is trivially carried through. This implies that particles 1 and 3 should be locked together. For ISR we don't need the q variable.

⟨*SF isr: types*⟩+≡
```
type, extends (sf_int_t) :: isr_t
   type(isr_data_t), pointer :: data => null ()
   real(default) :: x = 0
   real(default) :: xb= 0
 contains
 ⟨SF isr: isr: TBP⟩
end type isr_t
```

Type string: has to be here, but there is no string variable on which ISR depends. Hence, a dummy routine.

⟨*SF isr: isr: TBP*⟩≡
```
procedure :: type_string => isr_type_string
```

⟨*SF isr: procedures*⟩+≡
```
function isr_type_string (object) result (string)
  class(isr_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "ISR: e+ e- ISR spectrum"
  else
     string = "ISR: [undefined]"
  end if
end function isr_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF isr: isr: TBP*⟩+≡
```
procedure :: write => isr_write
```

⟨*SF isr: procedures*⟩+≡
```
subroutine isr_write (object, unit, testflag)
  class(isr_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
        write (u, "(3x,A," // FMT_17 // ")")  "xb=", object%xb
```

1181

```
         end if
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "ISR data: [undefined]"
      end if
   end subroutine isr_write
```

10.4.5   Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ were trivial. The ISR structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the x value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the input (random) number `r` to generate the momentum fraction `x` and the function value `f`. For numerical stability reasons, we also output `xb`, which is $\bar{x} = 1 - x$.

For the ISR structure function, the mapping Jacobian cancels the structure function (to order zero). We apply the cancellation explicitly, therefore both the Jacobian `f` and the zeroth-order value (see the `apply` method) are unity if mapping is turned on. If mapping is turned off, the Jacobian `f` includes the value of the (zeroth-order) structure function, and strongly peaked.

⟨*SF isr: isr: TBP*⟩+≡
```
    procedure :: complete_kinematics => isr_complete_kinematics
```

⟨*SF isr: procedures*⟩+≡
```
    subroutine isr_complete_kinematics (sf_int, x, f, r, rb, map)
      class(isr_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      real(default) :: eps
      eps = sf_int%data%eps
      if (map) then
         call map_power_1 (sf_int%xb, f, rb(1), eps)
      else
         sf_int%xb = rb(1)
         if (rb(1) > 0) then
            f = 1
         else
            f = 0
         end if
      end if
      sf_int%x = 1 - sf_int%xb
      x(1) = sf_int%x
      if (size (x) == 3)  x(2:3) = r(2:3)
      call sf_int%split_momentum (x, sf_int%xb)
      select case (sf_int%status)
      case (SF_FAILED_KINEMATICS)
         sf_int%x = 0
         sf_int%xb= 0
```

1182

```
          f = 0
     end select
   end subroutine isr_complete_kinematics
```

Overriding the default method: we compute the x array from the momentum configuration. In the specific case of ISR, we also set the internally stored x and $\bar{x}$ values, so they can be used in the following routine.

Note: the extraction of $\bar{x}$ is not numerically safe, but it can't be as long as the base `recover_x` isn't.

⟨*SF isr: isr: TBP*⟩+≡
```
   procedure :: recover_x => sf_isr_recover_x
```

⟨*SF isr: procedures*⟩+≡
```
   subroutine sf_isr_recover_x (sf_int, x, x_free)
     class(isr_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(out) :: x
     real(default), intent(inout), optional :: x_free
     call sf_int%base_recover_x (x, x_free)
     sf_int%x  = x(1)
     sf_int%xb = 1 - x(1)
   end subroutine sf_isr_recover_x
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

For extracting x, we rely on the stored $\bar{x}$ value, since the x value in the argument is likely imprecise. This means that either `complete_kinematics` or `recover_x` must be called first, for the current sampling point (but maybe another channel).

⟨*SF isr: isr: TBP*⟩+≡
```
   procedure :: inverse_kinematics => isr_inverse_kinematics
```

⟨*SF isr: procedures*⟩+≡
```
   subroutine isr_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
     class(isr_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(in) :: x
     real(default), intent(out) :: f
     real(default), dimension(:), intent(out) :: r
     real(default), dimension(:), intent(out) :: rb
     logical, intent(in) :: map
     logical, intent(in), optional :: set_momenta
     real(default) :: eps
     logical :: set_mom
     set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
     eps = sf_int%data%eps
     if (map) then
        call map_power_inverse_1 (sf_int%xb, f, rb(1), eps)
     else
        rb(1) = sf_int%xb
        if (rb(1) > 0) then
           f = 1
        else
           f = 0
```

1183

```
            end if
         end if
         r(1) = 1 - rb(1)
         if (size(r) == 3) then
            r(2:3) = x(2:3)
            rb(2:3)= 1 - r(2:3)
         end if
         if (set_mom) then
            call sf_int%split_momentum (x, sf_int%xb)
            select case (sf_int%status)
            case (SF_FAILED_KINEMATICS)
               r = 0
               rb= 0
               f = 0
            end select
         end if
      end subroutine isr_inverse_kinematics
```

⟨*SF isr: isr: TBP*⟩+≡
```
      procedure :: init => isr_init
```

⟨*SF isr: procedures*⟩+≡
```
      subroutine isr_init (sf_int, data)
        class(isr_t), intent(out) :: sf_int
        class(sf_data_t), intent(in), target :: data
        type(quantum_numbers_mask_t), dimension(3) :: mask
        integer, dimension(3) :: hel_lock
        type(polarization_t) :: pol
        type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel
        type(flavor_t) :: flv_photon
        type(quantum_numbers_t) :: qn_photon, qn
        type(state_iterator_t) :: it_hel
        real(default) :: m2
        integer :: i
        mask = new_quantum_numbers_mask (.false., .false., &
             mask_h = [.false., .true., .false.])
        hel_lock = [3, 0, 1]
        select type (data)
        type is (isr_data_t)
           m2 = data%mass**2
           call sf_int%base_init (mask, [m2], [0._default], [m2], &
                hel_lock = hel_lock)
           sf_int%data => data
           call flavor_init (flv_photon, PHOTON, data%model)
           call quantum_numbers_init (qn_photon, flv_photon)
           do i = 1, size (data%flv_in)
              call polarization_init_generic (pol, data%flv_in(i))
              call quantum_numbers_init (qn_fc(1), &
                   flv = data%flv_in(i), &
                   col = color_from_flavor (data%flv_in(i), 1))
              call state_iterator_init (it_hel, pol%state)
              do while (state_iterator_is_valid (it_hel))
                 qn_hel = state_iterator_get_quantum_numbers (it_hel)
                 qn = qn_hel(1) .merge. qn_fc(1)
```

```
            call interaction_add_state &
                 (sf_int%interaction_t, [qn, qn_photon, qn])
            call state_iterator_advance (it_hel)
          end do
          call polarization_final (pol)
       end do
       call interaction_freeze (sf_int%interaction_t)
       call sf_int%set_incoming ([1])
       call sf_int%set_radiated ([2])
       call sf_int%set_outgoing ([3])
       sf_int%status = SF_INITIAL
    end select
  end subroutine isr_init
```

10.4.6 ISR application

For ISR, we could in principle compute kinematics and function value in a single step. In order to be able to reweight matrix elements including structure functions we split kinematics and structure function calculation. The structure function works on a single beam, assuming that the input momentum has been set.

For the structure-function evaluation, we rely on the fact that the power mapping, which we apply in the kinematics method (if the `map` flag is set), has a Jacobian which is just the inverse lowest-order structure function. With mapping active, the two should cancel exactly.

After splitting momenta, we set the outgoing momenta on-shell. We choose to conserve momentum, so energy conservation may be violated.

⟨*SF isr: isr: TBP*⟩+≡
```
    procedure :: apply => isr_apply
```

⟨*SF isr: procedures*⟩+≡
```
    subroutine isr_apply (sf_int, scale)
      class(isr_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
      real(default) :: f, finv, x, xb, eps, rb
      real(default) :: log_x, log_xb, x_2
      real(default), parameter :: &
           & xmin = 0.0071405332973459283954 9879772019_default
      real(default), parameter :: &
           & zeta3 = 1.2020569031595942853997 3816151_default
      real(default), parameter :: &
           & g1 = 3._default / 4._default, &
           & g2 = (27 - 8*pi**2) / 96._default, &
           & g3 = (27 - 24*pi**2 + 128*zeta3) / 384._default
      associate (data => sf_int%data)
        eps = sf_int%data%eps
        x = sf_int%x
        xb = sf_int%xb
        call map_power_inverse_1 (xb, finv, rb, eps)
        if (finv > 0) then
           f = 1 / finv
        else
```

```
                  f = 0
            end if
            if (f > 0 .and. data%order > 0) then
                f = f * (1 + g1 * eps)
                x_2 = x*x
                if (rb>0)  f = f * (1 - (1-x_2) / (2 * rb))
                if (data%order > 1) then
                    f = f * (1 + g2 * eps**2)
                    if (rb>0 .and. xb>0 .and. x>xmin) then
                       log_x  = log_prec (x, xb)
                       log_xb = log_prec (xb, x)
                       f = f * (1 - ((1+3*x_2)*log_x + xb * (4*(1+x)*log_xb + 5 + x)) &
                             / ( 8 * rb) * eps)
                    end if
                    if (data%order > 2) then
                        f = f * (1 + g3 * eps**3)
                        if (rb > 0 .and. xb > 0 .and. x > xmin) then
                           f = f * (1 - ((1+x) * xb &
                                 * (6 * Li2(x) + 12 * log_xb**2 - 3 * pi**2) &
                                 + 1.5_default * (1 + 8*x + 3*x_2) * log_x &
                                 + 6 * (x+5) * xb * log_xb &
                                 + 12 * (1+x_2) * log_x * log_xb &
                                 - (1 + 7*x_2) * log_x**2 / 2 &
                                 + (39 - 24*x - 15*x_2) / 4) &
                                 / ( 48 * rb) * eps**2)
                        end if
                    end if
                end if
            end if
          end associate
          call interaction_set_matrix_element &
                (sf_int%interaction_t, cmplx (f, kind=default))
        sf_int%status = SF_EVALUATED
      end subroutine isr_apply
```

10.4.7 Unit tests

⟨*SF isr: public*⟩+≡
```
    public :: sf_isr_test
```

⟨*SF isr: tests*⟩≡
```
    subroutine sf_isr_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*SF isr: execute tests*⟩
```
    end subroutine sf_isr_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF isr: execute tests*⟩≡

```
        call test (sf_isr_1, "sf_isr_1", &
             "structure function configuration", &
             u, results)
```
⟨*SF isr: tests*⟩+≡
```
     subroutine sf_isr_1 (u)
       integer, intent(in) :: u
       type(os_data_t) :: os_data
       type(model_list_t) :: model_list
       type(model_t), pointer :: model
       type(pdg_array_t) :: pdg_in
       type(pdg_array_t), dimension(1) :: pdg_out
       integer, dimension(:), allocatable :: pdg1
       class(sf_data_t), allocatable :: data

       write (u, "(A)")  "* Test output: sf_isr_1"
       write (u, "(A)")  "*   Purpose: initialize and display &
            &test structure function data"
       write (u, "(A)")

       write (u, "(A)")  "* Create empty data object"
       write (u, "(A)")

       call os_data_init (os_data)
       call syntax_model_file_init ()
       call model_list%read_model (var_str ("QED"), &
            var_str ("QED.mdl"), os_data, model)
       pdg_in = ELECTRON

       allocate (isr_data_t :: data)
       call data%write (u)

       write (u, "(A)")
       write (u, "(A)")  "* Initialize"
       write (u, "(A)")

       select type (data)
       type is (isr_data_t)
          call data%init (model, pdg_in, 1./137._default, 10._default, &
               0.000511_default, order = 3, recoil = .false.)
       end select

       call data%write (u)

       write (u, "(A)")

       write (u, "(1x,A)")  "Outgoing particle codes:"
       call data%get_pdg_out (pdg_out)
       pdg1 = pdg_out(1)
       write (u, "(2x,99(1x,I0))")  pdg1

       call model_list%final ()
       call syntax_model_file_final ()

       write (u, "(A)")
```

1187

```
    write (u, "(A)")  "* Test output end: sf_isr_1"

  end subroutine sf_isr_1
```

Structure function without mapping

Direct ISR evaluation. This is the use case for a double-beam structure function. The parameter pair is mapped in the calling program.

⟨*SF isr: execute tests*⟩+≡
```
    call test (sf_isr_2, "sf_isr_2", &
         "no ISR mapping", &
         u, results)
```

⟨*SF isr: tests*⟩+≡
```
  subroutine sf_isr_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    type(pdg_array_t) :: pdg_in
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f, f_isr

    write (u, "(A)")  "* Test output: sf_isr_2"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list%read_model (var_str ("QED"), &
         var_str ("QED.mdl"), os_data, model)
    pdg_in = ELECTRON
    call flavor_init (flv, ELECTRON, model)

    call reset_interaction_counter ()

    allocate (isr_data_t :: data)
    select type (data)
    type is (isr_data_t)
       call data%init (model, pdg_in, 1./137._default, 500._default, &
            0.000511_default, order = 3, recoil = .false.)
    end select

    write (u, "(A)")  "* Initialize structure-function object"
```

```
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.9, no ISR mapping, &
     &collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.9_default
rb = 1 - r
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb

call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A)")
write (u, "(A,9(1x," // FMT_12 // "))")  "x =", x
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Invert kinematics"
write (u, "(A)")

call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"
write (u, "(A)")

call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure-function value, default order"
write (u, "(A)")
```

```
      f_isr = interaction_get_matrix_element (sf_int%interaction_t, 1)

      write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
      write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

      write (u, "(A)")
      write (u, "(A)")  "* Re-evaluate structure function, leading order"
      write (u, "(A)")

      select type (sf_int)
      type is (isr_t)
         sf_int%data%order = 0
      end select
      call sf_int%apply (scale = 100._default)
      f_isr = interaction_get_matrix_element (sf_int%interaction_t, 1)

      write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
      write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_isr_2"

    end subroutine sf_isr_2
```

Structure function with mapping

Apply the optimal ISR mapping. This is the use case for a single-beam structure
function.

⟨*SF isr: execute tests*⟩+≡
```
    call test (sf_isr_3, "sf_isr_3", &
         "ISR mapping", &
         u, results)
```
⟨*SF isr: tests*⟩+≡
```
    subroutine sf_isr_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
```

```
real(default) :: f, f_isr

write (u, "(A)")  "* Test output: sf_isr_3"
write (u, "(A)")  "*   Purpose: initialize and fill &
    &test structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("QED"), &
    var_str ("QED.mdl"), os_data, model)
call flavor_init (flv, ELECTRON, model)
pdg_in = ELECTRON

call reset_interaction_counter ()

allocate (isr_data_t :: data)
select type (data)
type is (isr_data_t)
   call data%init (model, pdg_in, 1./137._default, 500._default, &
       0.000511_default, order = 3, recoil = .false.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.7, with ISR mapping, &
    &collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.7_default
rb = 1 - r
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
```

```
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)

write (u, "(A)")
write (u, "(A,9(1x," // FMT_12 // "))")  "x =", x
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Invert kinematics"
write (u, "(A)")

call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"
write (u, "(A)")

call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure-function value, default order"
write (u, "(A)")

f_isr = interaction_get_matrix_element (sf_int%interaction_t, 1)

write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

write (u, "(A)")
write (u, "(A)")  "* Re-evaluate structure function, leading order"
write (u, "(A)")

select type (sf_int)
type is (isr_t)
   sf_int%data%order = 0
end select
call sf_int%apply (scale = 100._default)
f_isr = interaction_get_matrix_element (sf_int%interaction_t, 1)

write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
```

```
        write (u, "(A)")  "* Test output end: sf_isr_3"

      end subroutine sf_isr_3
```

Non-collinear ISR splitting

Construct and display a structure function object based on the ISR structure
function. We blank out numerical fluctuations for 32bit.

⟨*SF isr: execute tests*⟩+≡
```
      call test (sf_isr_4, "sf_isr_4", &
           "ISR non-collinear", &
           u, results)
```

⟨*SF isr: tests*⟩+≡
```
    subroutine sf_isr_4 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f, f_isr
      character(len=80) :: buffer
      integer :: u_scratch, iostat

      write (u, "(A)")  "* Test output: sf_isr_4"
      write (u, "(A)")  "*   Purpose: initialize and fill &
           &test structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QED"), &
           var_str ("QED.mdl"), os_data, model)
      call flavor_init (flv, ELECTRON, model)
      pdg_in = ELECTRON

      call reset_interaction_counter ()

      write (u, "(A)")
      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      allocate (isr_data_t :: data)
```

1193

```
select type (data)
type is (isr_data_t)
   call data%init (model, pdg_in, 1./137._default, 500._default, &
         0.000511_default, order = 3, recoil = .true.)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.25, with ISR mapping, "
write (u, "(A)")  "           non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
```

```
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "r =", r

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate ISR structure function"
      write (u, "(A)")

      call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
      call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
      call sf_int%apply (scale = 10._default)
      u_scratch = free_unit ()
      open (u_scratch, status="scratch", action = "readwrite")
      call sf_int%write (u_scratch, testflag = .true.)
      rewind (u_scratch)
      do
         read (u_scratch, "(A)", iostat=iostat) buffer
         if (iostat /= 0) exit
         if (buffer(1:27) == " P =    0.00000000E+00   9.57") then
            buffer = replace (buffer, 28, "XXXXXX")
         end if
         if (buffer(1:27) == " P =    0.00000000E+00  -9.57") then
            buffer = replace (buffer, 28, "XXXXXX")
         end if
         write (u, "(A)") buffer
      end do
      close (u_scratch)

      write (u, "(A)")
      write (u, "(A)")  "* Structure-function value"
      write (u, "(A)")

      f_isr = interaction_get_matrix_element (sf_int%interaction_t, 1)

      write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
      write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_isr_4"

    end subroutine sf_isr_4
```

Structure function pair with mapping

Apply the ISR mapping for a ISR pair. structure function.

⟨*SF isr: execute tests*⟩+≡

```
      call test (sf_isr_5, "sf_isr_5", &
           "ISR pair mapping", &
           u, results)
```

⟨*SF isr: tests*⟩+≡

```
     subroutine sf_isr_5 (u)
       integer, intent(in) :: u
       type(os_data_t) :: os_data
       type(model_list_t) :: model_list
       type(model_t), pointer :: model
       type(flavor_t) :: flv
       type(pdg_array_t) :: pdg_in
       class(sf_data_t), allocatable, target :: data
       class(sf_mapping_t), allocatable :: mapping
       class(sf_int_t), dimension(:), allocatable :: sf_int
       type(vector4_t), dimension(2) :: k
       real(default) :: E, f_map
       real(default), dimension(:), allocatable :: p, pb, r, rb, x
       real(default), dimension(2) :: f, f_isr
       integer :: i

       write (u, "(A)")  "* Test output: sf_isr_5"
       write (u, "(A)")  "*   Purpose: initialize and fill &
            &test structure function object"
       write (u, "(A)")

       write (u, "(A)")  "* Initialize configuration data"
       write (u, "(A)")

       call os_data_init (os_data)
       call syntax_model_file_init ()
       call model_list%read_model (var_str ("QED"), &
            var_str ("QED.mdl"), os_data, model)
       call flavor_init (flv, ELECTRON, model)
       pdg_in = ELECTRON

       call reset_interaction_counter ()

       allocate (isr_data_t :: data)
       select type (data)
       type is (isr_data_t)
          call data%init (model, pdg_in, 1./137._default, 500._default, &
               0.000511_default, order = 3, recoil = .false.)
       end select

       allocate (sf_ip_mapping_t :: mapping)
       select type (mapping)
       type is (sf_ip_mapping_t)
          select type (data)
          type is (isr_data_t)
             call mapping%init (eps = data%eps)
          end select
          call mapping%set_index (1, 1)
          call mapping%set_index (2, 2)
       end select
```

1196

```
call mapping%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (isr_t :: sf_int (2))

do i = 1, 2
   call sf_int(i)%init (data)
   call sf_int(i)%set_beam_index ([i])
end do

write (u, "(A)")  "* Initialize incoming momenta with E=500"
write (u, "(A)")
E = 500
k(1) = vector4_moving (E,   sqrt (E**2 - flavor_get_mass (flv)**2), 3)
k(2) = vector4_moving (E, - sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
do i = 1, 2
   call vector4_write (k(i), u)
   call sf_int(i)%seed_kinematics (k(i:i))
end do

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for p=[0.7,0.4], collinear"
write (u, "(A)")

allocate (p (2 * data%get_n_par ()))
allocate (pb(size (p)))
allocate (r (size (p)))
allocate (rb(size (p)))
allocate (x (size (p)))

p = [0.7_default, 0.4_default]
pb= 1 - p
call mapping%compute (r, rb, f_map, p, pb)

write (u, "(A,9(1x," // FMT_12 // "))")  "p =", p
write (u, "(A,9(1x," // FMT_12 // "))")  "pb=", pb
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "fm=", f_map

do i = 1, 2
   call sf_int(i)%complete_kinematics (x(i:i), f(i), r(i:i), rb(i:i), &
        map=.false.)
end do

write (u, "(A)")
write (u, "(A,9(1x," // FMT_12 // "))")  "x =", x
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f
```

```
write (u, "(A)")
write (u, "(A)")  "* Invert kinematics"
write (u, "(A)")

do i = 1, 2
   call sf_int(i)%inverse_kinematics (x(i:i), f(i), r(i:i), rb(i:i), &
        map=.false.)
end do
call mapping%inverse (r, rb, f_map, p, pb)

write (u, "(A,9(1x," // FMT_12 // "))")  "p =", p
write (u, "(A,9(1x," // FMT_12 // "))")  "pb=", pb
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "fm=", f_map

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"

call sf_int(1)%apply (scale = 100._default)
call sf_int(2)%apply (scale = 100._default)

write (u, "(A)")
write (u, "(A)")  "* Structure function #1"
write (u, "(A)")
call sf_int(1)%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure function #2"
write (u, "(A)")
call sf_int(2)%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure-function value, default order"
write (u, "(A)")

do i = 1, 2
   f_isr(i) = interaction_get_matrix_element (sf_int(i)%interaction_t, 1)
end do

write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", &
     product (f_isr)
write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", &
     product (f_isr * f) * f_map

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

do i = 1, 2
   call sf_int(i)%final ()
end do
call model_list%final ()
call syntax_model_file_final ()
```

```
     write (u, "(A)")
     write (u, "(A)")  "* Test output end: sf_isr_5"

end subroutine sf_isr_5
```

10.5 EPA

⟨sf_epa.f90⟩≡
  ⟨*File header*⟩

```
module sf_epa
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use constants, only: pi !NODEP!
```
  ⟨*Use file utils*⟩
```
    use limits, only: FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use pdg_arrays
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base
```

  ⟨*Standard module head*⟩

  ⟨*SF epa: public*⟩

  ⟨*SF epa: parameters*⟩

  ⟨*SF epa: types*⟩

```
  contains
```

  ⟨*SF epa: procedures*⟩

  ⟨*SF epa: tests*⟩

```
  end module sf_epa
```

10.5.1 Physics

The EPA structure function for a photon inside an (elementary) particle p with energy E, mass m and charge q_p (e.g., electron) is given by ($\bar{x} \equiv 1 - x$)

$$
f(x) = \frac{\alpha}{\pi} q_p^2 \frac{1}{x} \left[\left(\bar{x} + \frac{x^2}{2} \right) \ln \frac{Q_{\max}^2}{Q_{\min}^2} \right.
$$

$$
\left. - \left(1 - \frac{x}{2} \right)^2 \ln \frac{x^2 + \frac{Q_{\max}^2}{E^2}}{x^2 + \frac{Q_{\min}^2}{E^2}} - x^2 \frac{m^2}{Q_{\min}^2} \left(1 - \frac{Q_{\min}^2}{Q_{\max}^2} \right) \right]. \quad (10.52)
$$

If no explicit Q bounds are provided, the kinematical bounds are

$$-Q_{\max}^2 = t_0 = -2\bar{x}(E^2 + p\bar{p}) + 2m^2 \approx -4\bar{x}E^2,$$
(10.53)

$$-Q_{\min}^2 = t_1 = -2\bar{x}(E^2 - p\bar{p}) + 2m^2 \approx -\frac{x^2}{\bar{x}}m^2.$$
(10.54)

The second and third terms in (10.52) are negative definite (and subleading). Noting that $\bar{x}+x^2/2$ is bounded between $1/2$ and 1, we derive that $f(x)$ is always smaller than

$$\bar{f}(x) = \frac{\alpha}{\pi}\, q_p^2\, \frac{L - 2\ln x}{x} \qquad \text{where} \qquad L = \ln\frac{\min(4E_{\max}^2, Q_{\max}^2)}{\max(m^2, Q_{\min}^2)},$$
(10.55)

where we allow for explicit Q bounds that narrow the kinematical range. Therefore, we generate this distribution:

$$\int_{x_0}^{x_1} dx\, \bar{f}(x) = C(x_0, x_1) \int_0^1 dx'$$
(10.56)

We set

$$\ln x = \frac{1}{2}\left\{L - \sqrt{L^2 - 4\left[x'\ln x_1(L - \ln x_1) + \bar{x}'\ln x_0(L - \ln x_0)\right]}\right\}$$
(10.57)

such that $x(0) = x_0$ and $x(1) = x_1$ and

$$\frac{dx}{dx'} = \left(\frac{\alpha}{\pi}q_p^2\right)^{-1} x\frac{C(x_0, x_1)}{L - 2\ln x}$$
(10.58)

with

$$C(x_0, x_1) = \frac{\alpha}{\pi}q_p^2\,\left[\ln x_1(L - \ln x_1) - \ln x_0(L - \ln x_0)\right]$$
(10.59)

such that (10.56) is satisfied. Finally, we have

$$\int_{x_0}^{x_1} dx\, f(x) = C(x_0, x_1) \int_0^1 dx'\, \frac{f(x(x'))}{\bar{f}(x(x'))}$$
(10.60)

where x' is calculated from x via (10.57).

The structure of the mapping is most obvious from:

$$x'(x) = \frac{\log x(L - \log x) - \log x_0(L - \log x_0)}{\log x_1(L - \log x_1) - \log x_0(L - \log x_0)}\ .$$
(10.61)

10.5.2 The EPA data block

The EPA parameters are: α, $E_{\max}$, m, $Q_{\min}$, and $x_{\min}$. Instead of m we can use the incoming particle PDG code as input; from this we can deduce the mass and charge.

Internally we store in addition $C_{0/1} = \frac{\alpha}{\pi}q_e^2 \ln x_{0/1}(L - \ln x_{0/1})$, the c.m. energy squared and the incoming particle mass.

⟨*SF epa: public*⟩≡
```
public :: epa_data_t
```

⟨SF epa: types⟩≡

```
type, extends(sf_data_t) :: epa_data_t
   private
   type(model_t), pointer :: model => null ()
   type(flavor_t), dimension(:), allocatable :: flv_in
   real(default) :: alpha
   real(default) :: x_min
   real(default) :: x_max
   real(default) :: q_min
   real(default) :: q_max
   real(default) :: E_max
   real(default) :: mass
   real(default) :: log
   real(default) :: a
   real(default) :: c0
   real(default) :: c1
   real(default) :: dc
   integer :: error = NONE
   logical :: recoil = .false.
 contains
 ⟨SF epa: epa data: TBP⟩
end type epa_data_t
```

Error codes

⟨SF epa: parameters⟩≡

```
integer, parameter :: NONE = 0
integer, parameter :: ZERO_QMIN = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: ZERO_XMIN = 3
integer, parameter :: MASS_MIX = 4
integer, parameter :: NO_EPA = 5
```

⟨SF epa: epa data: TBP⟩≡

```
procedure :: init => epa_data_init
```

⟨SF epa: procedures⟩≡

```
subroutine epa_data_init &
    (data, model, pdg_in, alpha, x_min, q_min, E_max, mass, recoil)
  class(epa_data_t), intent(inout) :: data
  type(model_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  real(default), intent(in) :: alpha, x_min, q_min, E_max
  real(default), intent(in), optional :: mass
  logical, intent(in), optional :: recoil
  integer :: n_flv, i
  data%model => model
  n_flv = pdg_array_get_length (pdg_in)
  allocate (data%flv_in (n_flv))
  do i = 1, n_flv
     call flavor_init (data%flv_in(i), pdg_array_get (pdg_in, i), model)
  end do
  data%alpha = alpha
  data%E_max = E_max
  data%x_min = x_min
  data%x_max = 1
```

```
          if (data%x_min == 0) then
              data%error = ZERO_XMIN;  return
          end if
          data%q_min = q_min
          data%q_max = 2 * data%E_max
          select case (char (data%model%get_name ()))
          case ("QCD","Test")
              data%error = NO_EPA;  return
          end select
          if (present (recoil)) then
              data%recoil = recoil
          end if
          if (present (mass)) then
              data%mass = mass
          else
              data%mass = flavor_get_mass (data%flv_in(1))
              if (any (flavor_get_mass (data%flv_in) /= data%mass)) then
                  data%error = MASS_MIX;  return
              end if
          end if
          if (max (data%mass, data%q_min) == 0) then
              data%error = ZERO_QMIN;  return
          else if (max (data%mass, data%q_min) >= data%E_max) then
              data%error = Q_MAX_TOO_SMALL;  return
          end if
          data%log = log (4 * (data%E_max / max (data%mass, data%q_min)) ** 2 )
          data%a   = data%alpha / pi
          data%c0 = log (data%x_min) * (data%log - log (data%x_min))
          data%c1 = log (data%x_max) * (data%log - log (data%x_max))
          data%dc = data%c1 - data%c0
      end subroutine epa_data_init
```

Handle error conditions. Should always be done after initialization, unless we
are sure everything is ok.

⟨*SF epa: epa data: TBP*⟩+≡
```
      procedure :: check => epa_data_check
```

⟨*SF epa: procedures*⟩+≡
```
      subroutine epa_data_check (data)
        class(epa_data_t), intent(in) :: data
        select case (data%error)
        case (NO_EPA)
          call msg_fatal ("EPA structure function not available for model " &
                // char (data%model%get_name ()) // ".")
        case (ZERO_QMIN)
          call msg_fatal ("EPA: Particle mass is zero")
        case (Q_MAX_TOO_SMALL)
          call msg_fatal ("EPA: Particle mass exceeds Qmax")
        case (ZERO_XMIN)
          call msg_fatal ("EPA: x_min must be larger than zero")
        case (MASS_MIX)
          call msg_fatal ("EPA: incoming particle masses must be uniform")
        end select
      end subroutine epa_data_check
```

Output

⟨*SF epa: epa data: TBP*⟩+≡
```
procedure :: write => epa_data_write
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_data_write (data, unit, verbose)
  class(epa_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "EPA data:"
  if (allocated (data%flv_in)) then
     write (u, "(3x,A)", advance="no") "  flavor =  "
     do i = 1, size (data%flv_in)
        if (i > 1)  write (u, "(',',1x)", advance="no")
        call flavor_write (data%flv_in(i), u)
     end do
     write (u, *)
     write (u, "(3x,A," // FMT_19 // ")") "  alpha  = ", data%alpha
     write (u, "(3x,A," // FMT_19 // ")") "  x_min  = ", data%x_min
     write (u, "(3x,A," // FMT_19 // ")") "  x_max  = ", data%x_max
     write (u, "(3x,A," // FMT_19 // ")") "  q_min  = ", data%q_min
     write (u, "(3x,A," // FMT_19 // ")") "  q_max  = ", data%q_max
     write (u, "(3x,A," // FMT_19 // ")") "  E_max  = ", data%e_max
     write (u, "(3x,A," // FMT_19 // ")") "  mass   = ", data%mass
     write (u, "(3x,A," // FMT_19 // ")") "  a      = ", data%a
     write (u, "(3x,A," // FMT_19 // ")") "  c0     = ", data%c0
     write (u, "(3x,A," // FMT_19 // ")") "  c1     = ", data%c1
     write (u, "(3x,A," // FMT_19 // ")") "  log    = ", data%log
     write (u, "(3x,A,L2)")       "  recoil = ", data%recoil
  else
     write (u, "(3x,A)") "[undefined]"
  end if
end subroutine epa_data_write
```

The number of kinematic parameters.

⟨*SF epa: epa data: TBP*⟩+≡
```
procedure :: get_n_par => epa_data_get_n_par
```

⟨*SF epa: procedures*⟩+≡
```
function epa_data_get_n_par (data) result (n)
  class(epa_data_t), intent(in) :: data
  integer :: n
  if (data%recoil) then
     n = 3
  else
     n = 1
  end if
end function epa_data_get_n_par
```

1204

Return the outgoing particles PDG codes. The outgoing particle is always the photon while the radiated particle is identical to the incoming one.

⟨*SF epa: epa data: TBP*⟩+≡
```
procedure :: get_pdg_out => epa_data_get_pdg_out
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_data_get_pdg_out (data, pdg_out)
  class(epa_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  pdg_out(1) = PHOTON
end subroutine epa_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF epa: epa data: TBP*⟩+≡
```
procedure :: allocate_sf_int => epa_data_allocate_sf_int
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_data_allocate_sf_int (data, sf_int)
  class(epa_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (epa_t :: sf_int)
end subroutine epa_data_allocate_sf_int
```

10.5.3 The EPA object

The `epa_t` data type is a $1 \to 2$ interaction. We should be able to handle several flavors in parallel, since EPA is not necessarily applied immediately after beam collision: Photons may be radiated from quarks. In that case, the partons are massless and $q_{\min}$ applies instead, so we do not need to generate several kinematical configurations in parallel.

The squared charge values multiply the matrix elements, depending on the flavour. We scan the interaction after building it, so we have the correct assignments.

The particles are ordered as (incoming, radiated, photon), where the photon initiates the hard interaction.

We generate an unpolarized photon and transfer initial polarization to the radiated parton. Color is transferred in the same way.

⟨*SF epa: types*⟩+≡
```
type, extends (sf_int_t) :: epa_t
  type(epa_data_t), pointer :: data => null ()
  real(default) :: x  = 0
  real(default) :: xb = 0
  real(default) :: E  = 0
  real(default), dimension(:), allocatable :: charge2
 contains
 ⟨SF epa: epa: TBP⟩
end type epa_t
```

Type string: has to be here, but there is no string variable on which EPA depends. Hence, a dummy routine.

⟨*SF epa: epa: TBP*⟩≡
```
procedure :: type_string => epa_type_string
```

⟨*SF epa: procedures*⟩+≡
```
function epa_type_string (object) result (string)
  class(epa_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "EPA: equivalent photon approx."
  else
     string = "EPA: [undefined]"
  end if
end function epa_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF epa: epa: TBP*⟩+≡
```
procedure :: write => epa_write
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_write (object, unit, testflag)
  class(epa_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
           write (u, "(3x,A," // FMT_17 // ")")  "E =", object%E
        end if
     end if
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "EPA data: [undefined]"
  end if
end subroutine epa_write
```

Prepare the interaction object. We have to construct transition matrix elements for all flavor and helicity combinations.

⟨*SF epa: epa: TBP*⟩+≡
```
procedure :: init => epa_init
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_init (sf_int, data)
  class(epa_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: hel_lock
  type(polarization_t) :: pol
```

1206

```
      type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel
      type(flavor_t) :: flv_photon
      type(quantum_numbers_t) :: qn_photon, qn
      type(state_iterator_t) :: it_hel
      integer :: i
      mask = new_quantum_numbers_mask (.false., .false., &
          mask_h = [.false., .false., .true.])
      hel_lock = [2, 1, 0]
      select type (data)
      type is (epa_data_t)
         call sf_int%base_init (mask, [data%mass**2], &
             [data%mass**2], [0._default], hel_lock = hel_lock)
         sf_int%data => data
         call flavor_init (flv_photon, PHOTON, data%model)
         call quantum_numbers_init (qn_photon, flv_photon)
         do i = 1, size (data%flv_in)
            call polarization_init_generic (pol, data%flv_in(i))
            call quantum_numbers_init (qn_fc(1), &
                flv = data%flv_in(i), &
                col = color_from_flavor (data%flv_in(i), 1))
            call state_iterator_init (it_hel, pol%state)
            do while (state_iterator_is_valid (it_hel))
               qn_hel = state_iterator_get_quantum_numbers (it_hel)
               qn = qn_hel(1) .merge. qn_fc(1)
               call interaction_add_state (sf_int%interaction_t, &
                   [qn, qn, qn_photon])
               call state_iterator_advance (it_hel)
            end do
            call polarization_final (pol)
         end do
         call interaction_freeze (sf_int%interaction_t)
         call sf_int%set_incoming ([1])
         call sf_int%set_radiated ([2])
         call sf_int%set_outgoing ([3])
      end select
    end subroutine epa_init
```

Prepare the charge array. This is separate from the previous routine since the state matrix may be helicity-contracted.

⟨*SF epa: epa: TBP*⟩+≡
```
    procedure :: setup_constants => epa_setup_constants
```

⟨*SF epa: procedures*⟩+≡
```
    subroutine epa_setup_constants (sf_int)
      class(epa_t), intent(inout) :: sf_int
      type(state_iterator_t) :: it
      integer :: i, n_me
      n_me = interaction_get_n_matrix_elements (sf_int%interaction_t)
      allocate (sf_int%charge2 (n_me))
      call state_iterator_init (it, &
          interaction_get_state_matrix_ptr (sf_int%interaction_t))
      do while (state_iterator_is_valid (it))
         i = state_iterator_get_me_index (it)
         sf_int%charge2(i) = &
```

```
            flavor_get_charge (state_iterator_get_flavor (it, 1)) ** 2
       call state_iterator_advance (it)
     end do
     sf_int%status = SF_INITIAL
  end subroutine epa_setup_constants
```

10.5.4 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

The EPA structure function allows for a straightforward mapping of the unit interval. The x value is transformed, and the mapped structure function becomes unity at its upper boundary.

The structure function implementation applies the above mapping to the input (random) number `r` to generate the momentum fraction `x` and the function value `f`. For numerical stability reasons, we also output `xb`, which is $\bar{x} = 1 - x$.

⟨*SF epa: epa: TBP*⟩+≡
```
    procedure :: complete_kinematics => epa_complete_kinematics
```

⟨*SF epa: procedures*⟩+≡
```
    subroutine epa_complete_kinematics (sf_int, x, f, r, rb, map)
      class(epa_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      real(default) :: xb1
      real(default) :: delta, sqrt_delta, lx
      if (map) then
         associate (data => sf_int%data)
           delta = data%log ** 2 -  4 * (r(1) * data%c1 + rb(1) * data%c0)
           if (delta > 0) then
              sqrt_delta = sqrt (delta)
              lx = (data%log - sqrt_delta) / 2
           else
              sf_int%status = SF_FAILED_KINEMATICS
              f = 0
              return
           end if
           x(1) = exp (lx)
           f = x(1) * data%dc / sqrt_delta
         end associate
      else
         x(1) = r(1)
         if (sf_int%data%x_min < x(1) .and. x(1) < sf_int%data%x_max) then
            f = 1
         else
            sf_int%status = SF_FAILED_KINEMATICS
            f = 0
            return
         end if
```

1208

```
      end if
      xb1 = 1 - x(1)
      if (size(x) == 3)  x(2:3) = r(2:3)
      call sf_int%split_momentum (x, xb1)
      select case (sf_int%status)
      case (SF_DONE_KINEMATICS)
         sf_int%x = x(1)
         sf_int%xb= xb1
         sf_int%E  = &
              energy (interaction_get_momentum (sf_int%interaction_t, 1))
      case (SF_FAILED_KINEMATICS)
         sf_int%x = 0
         sf_int%xb= 0
         f = 0
      end select
    end subroutine epa_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF epa: epa: TBP*⟩+≡
```
    procedure :: inverse_kinematics => epa_inverse_kinematics
```

⟨*SF epa: procedures*⟩+≡
```
    subroutine epa_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
      class(epa_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      real(default) :: lx, delta, sqrt_delta, c
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      if (map) then
         associate (data => sf_int%data)
           lx = log (x(1))
           sqrt_delta = data%log - 2 * lx
           delta = sqrt_delta ** 2
           c = (data%log ** 2 - delta) / 4
           r (1) = (c - data%c0) / data%dc
           rb(1) = (data%c1 - c) / data%dc
           f = x(1) * data%dc / sqrt_delta
         end associate
      else
         r (1) = x(1)
         rb(1) = 1 - x(1)
         if (sf_int%data%x_min < x(1) .and. x(1) < sf_int%data%x_max) then
            f = 1
         else
            f = 0
         end if
      end if
```

1209

```
        if (size(r) == 3) then
            r (2:3) = x(2:3)
            rb(2:3) = 1 - x(2:3)
        end if
        if (set_mom) then
            call sf_int%split_momentum (x, sf_int%xb)
            select case (sf_int%status)
            case (SF_DONE_KINEMATICS)
                sf_int%x  = x(1)
                sf_int%xb = 1 - x(1)
                sf_int%E  = &
                    energy (interaction_get_momentum (sf_int%interaction_t, 1))
            case (SF_FAILED_KINEMATICS)
                sf_int%x = 0
                f = 0
            end select
        end if
    end subroutine epa_inverse_kinematics
```

Overriding the default method: we compute the x array from the momentum configuration. In the specific case of EPA, we also set the internally stored x and $\bar{x}$ values, so they can be used in the following routine.

Note: the extraction of $\bar{x}$ is not numerically safe, but it can't be as long as the base `recover_x` isn't.

⟨*SF epa: epa: TBP*⟩+≡
```
    procedure :: recover_x => sf_epa_recover_x
```

⟨*SF epa: procedures*⟩+≡
```
    subroutine sf_epa_recover_x (sf_int, x, x_free)
      class(epa_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(inout), optional :: x_free
      call sf_int%base_recover_x (x, x_free)
      sf_int%x  = x(1)
      sf_int%xb = 1 - x(1)
    end subroutine sf_epa_recover_x
```

10.5.5   EPA application

For EPA, we can in principle compute kinematics and function value in a single step. In order to be able to reweight events, kinematics and structure function application are separated. This function works on a single beam, assuming that the input momentum has been set. We need three random numbers as input: one for x, and two for the polar and azimuthal angles. Alternatively, for the no-recoil case, we can skip p_T generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated x values are identical.

⟨*SF epa: epa: TBP*⟩+≡
```
    procedure :: apply => epa_apply
```

⟨*SF epa: procedures*⟩+≡
```
    subroutine epa_apply (sf_int, scale)
      class(epa_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
      real(default) :: x, xb, qminsq, qmaxsq, f, E
      associate (data => sf_int%data)
        x = sf_int%x
        xb= sf_int%xb
        E = sf_int%E
        qminsq = max (x ** 2 / xb * data%mass ** 2, data%q_min ** 2)
        qmaxsq = min (4 * xb * E ** 2, data%q_max ** 2)
        if (qminsq < qmaxsq) then
           f = data%a / x &
                 * ((xb + x ** 2 / 2) * log (qmaxsq / qminsq) &
                 - (1 - x / 2) ** 2 &
                 * log ((x**2 + qmaxsq / E ** 2) / (x**2 + qminsq / E ** 2)) &
                 - x ** 2 * data%mass ** 2 / qminsq * (1 - qminsq / qmaxsq))
        else
           f = 0
        end if
        call interaction_set_matrix_element &
             (sf_int%interaction_t, cmplx (f, kind=default) * sf_int%charge2)
      end associate
      sf_int%status = SF_EVALUATED
    end subroutine epa_apply
```

10.5.6   Unit tests

⟨*SF epa: public*⟩+≡
```
    public :: sf_epa_test
```

⟨*SF epa: tests*⟩≡
```
    subroutine sf_epa_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*SF epa: execute tests*⟩
```
    end subroutine sf_epa_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF epa: execute tests*⟩≡
```
    call test (sf_epa_1, "sf_epa_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF epa: tests*⟩+≡
```
    subroutine sf_epa_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
```

```
    type(pdg_array_t) :: pdg_in
    type(pdg_array_t), dimension(1) :: pdg_out
    integer, dimension(:), allocatable :: pdg1
    class(sf_data_t), allocatable :: data

    write (u, "(A)")  "* Test output: sf_epa_1"
    write (u, "(A)")  "*   Purpose: initialize and display &
         &test structure function data"
    write (u, "(A)")

    write (u, "(A)")  "* Create empty data object"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list%read_model (var_str ("QED"), &
         var_str ("QED.mdl"), os_data, model)
    pdg_in = ELECTRON

    allocate (epa_data_t :: data)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    select type (data)
    type is (epa_data_t)
       call data%init (model, pdg_in, 1./137._default, 0.01_default, &
            10._default, 50._default, 0.000511_default, recoil = .false.)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_epa_1"

  end subroutine sf_epa_1
```

Test and probe structure function

Construct and display a structure function object based on the EPA structure function.

⟨*SF epa: execute tests*⟩+≡

```
call test (sf_epa_2, "sf_epa_2", &
    "structure function instance", &
    u, results)
```

⟨*SF epa: tests*⟩+≡

```
subroutine sf_epa_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_epa_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
      &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("QED"), &
      var_str ("QED.mdl"), os_data, model)
  call flavor_init (flv, ELECTRON, model)
  pdg_in = ELECTRON

  call reset_interaction_counter ()

  allocate (epa_data_t :: data)
  select type (data)
  type is (epa_data_t)
     call data%init (model, pdg_in, 1./137._default, 0.01_default, &
          10._default, 50._default, 0.000511_default, recoil = .false.)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])
  call sf_int%setup_constants ()

  write (u, "(A)")  "* Initialize incoming momentum with E=500"
  write (u, "(A)")
```

1213

```
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, no EPA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false., &
     set_momenta=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Evaluate EPA structure function"
write (u, "(A)")

call sf_int%apply (scale = 100._default)
call sf_int%write (u)
```

```
      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_epa_2"

   end subroutine sf_epa_2
```

Standard mapping

Construct and display a structure function object based on the EPA structure function, applying the standard single-particle mapping.

⟨*SF epa: execute tests*⟩+≡
```
   call test (sf_epa_3, "sf_epa_3", &
        "apply mapping", &
        u, results)
```

⟨*SF epa: tests*⟩+≡
```
   subroutine sf_epa_3 (u)
     integer, intent(in) :: u
     type(os_data_t) :: os_data
     type(model_list_t) :: model_list
     type(model_t), pointer :: model
     type(flavor_t) :: flv
     type(pdg_array_t) :: pdg_in
     class(sf_data_t), allocatable, target :: data
     class(sf_int_t), allocatable :: sf_int
     type(vector4_t) :: k
     type(vector4_t), dimension(2) :: q
     real(default) :: E
     real(default), dimension(:), allocatable :: r, rb, x
     real(default) :: f

     write (u, "(A)")  "* Test output: sf_epa_3"
     write (u, "(A)")  "*   Purpose: initialize and fill &
          &test structure function object"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize configuration data"
     write (u, "(A)")

     call os_data_init (os_data)
     call syntax_model_file_init ()
     call model_list%read_model (var_str ("QED"), &
          var_str ("QED.mdl"), os_data, model)
     call flavor_init (flv, ELECTRON, model)
     pdg_in = ELECTRON

     call reset_interaction_counter ()
```

1215

```
allocate (epa_data_t :: data)
select type (data)
type is (epa_data_t)
   call data%init (model, pdg_in, 1./137._default, 0.01_default, &
        10._default, 50._default, 0.000511_default, recoil = .false.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, with EPA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()
```

```
          call sf_int%seed_kinematics ([k])
          call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
          call sf_int%recover_x (x)
          call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
               set_momenta=.true.)

          write (u, "(A,9(1x,F10.7))")  "r =", r
          write (u, "(A,9(1x,F10.7))")  "rb=", rb
          write (u, "(A,9(1x,F10.7))")  "x =", x
          write (u, "(A,9(1x,F10.7))")  "f =", f

          write (u, "(A)")
          write (u, "(A)")  "* Evaluate EPA structure function"
          write (u, "(A)")

          call sf_int%apply (scale = 100._default)
          call sf_int%write (u)

          write (u, "(A)")
          write (u, "(A)")  "* Cleanup"

          call sf_int%final ()
          call model_list%final ()
          call syntax_model_file_final ()

          write (u, "(A)")
          write (u, "(A)")  "* Test output end: sf_epa_3"

        end subroutine sf_epa_3
```

Non-collinear case

Construct and display a structure function object based on the EPA structure
function.

⟨*SF epa: execute tests*⟩+≡
```
      call test (sf_epa_4, "sf_epa_4", &
           "non-collinear", &
           u, results)
```
⟨*SF epa: tests*⟩+≡
```
      subroutine sf_epa_4 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_list_t) :: model_list
        type(model_t), pointer :: model
        type(flavor_t) :: flv
        type(pdg_array_t) :: pdg_in
        class(sf_data_t), allocatable, target :: data
        class(sf_int_t), allocatable :: sf_int
        type(vector4_t) :: k
        type(vector4_t), dimension(2) :: q
        real(default) :: E, m
        real(default), dimension(:), allocatable :: r, rb, x
```

```
      real(default) :: f

write (u, "(A)")  "* Test output: sf_epa_4"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &test structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("QED"), &
     var_str ("QED.mdl"), os_data, model)
call flavor_init (flv, ELECTRON, model)
pdg_in = ELECTRON

call reset_interaction_counter ()

allocate (epa_data_t :: data)
select type (data)
type is (epa_data_t)
   call data%init (model, pdg_in, 1./137._default, 0.01_default, &
        10._default, 50._default, 5.0_default, recoil = .true.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=500, me = 5 GeV"
write (u, "(A)")
E = 500
m = 5
k = vector4_moving (E, sqrt (E**2 - m**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.5/0.5/0.25, with EPA mapping, "
write (u, "(A)")  "             non-coll., keeping energy, me = 5 GeV"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
rb = 1 - r
```

1218

```
      sf_int%on_shell_mode = KEEP_ENERGY
      call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
      call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

      write (u, "(A,9(1x,F10.7))")  "r =", r
      write (u, "(A,9(1x,F10.7))")  "rb=", rb
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Recover x and r from momenta"
      write (u, "(A)")

      q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
      call sf_int%final ()
      deallocate (sf_int)

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
      call sf_int%set_beam_index ([1])
      call sf_int%setup_constants ()

      call sf_int%seed_kinematics ([k])
      call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
      call sf_int%recover_x (x)
      call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
           set_momenta=.true.)
      call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

      write (u, "(A,9(1x,F10.7))")  "r =", r
      write (u, "(A,9(1x,F10.7))")  "rb=", rb
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate EPA structure function"
      write (u, "(A)")

      call sf_int%apply (scale = 100._default)
      call sf_int%write (u, testflag = .true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_epa_4"

    end subroutine sf_epa_4
```

Structure function for multiple flavors

Construct and display a structure function object based on the EPA structure function. The incoming state has multiple particles with non-uniform charge.

⟨*SF epa: execute tests*⟩+≡
```
call test (sf_epa_5, "sf_epa_5", &
     "multiple flavors", &
     u, results)
```

⟨*SF epa: tests*⟩+≡
```
subroutine sf_epa_5 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_epa_5"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("SM"), &
       var_str ("SM.mdl"), os_data, model)
  call flavor_init (flv, 1, model)
  pdg_in = [1, 2, -1, -2]

  call reset_interaction_counter ()

  allocate (epa_data_t :: data)
  select type (data)
  type is (epa_data_t)
     call data%init (model, pdg_in, 1./137._default, 0.01_default, &
          10._default, 50._default, 0.000511_default, recoil = .false.)
     call data%check ()
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])
```

1220

```
   call sf_int%setup_constants ()

   write (u, "(A)")  "* Initialize incoming momentum with E=500"
   write (u, "(A)")
   E = 500
   k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
   call pacify (k, 1e-10_default)
   call vector4_write (k, u)
   call sf_int%seed_kinematics ([k])

   write (u, "(A)")
   write (u, "(A)")  "* Set kinematics for r=0.4, no EPA mapping, collinear"
   write (u, "(A)")

   allocate (r (data%get_n_par ()))
   allocate (rb(size (r)))
   allocate (x (size (r)))

   r = 0.4_default
   rb = 1 - r
   call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

   write (u, "(A,9(1x,F10.7))")  "r =", r
   write (u, "(A,9(1x,F10.7))")  "rb=", rb
   write (u, "(A,9(1x,F10.7))")  "x =", x
   write (u, "(A,9(1x,F10.7))")  "f =", f

   write (u, "(A)")
   write (u, "(A)")  "* Evaluate EPA structure function"
   write (u, "(A)")

   call sf_int%apply (scale = 100._default)
   call sf_int%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call sf_int%final ()
   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: sf_epa_5"

end subroutine sf_epa_5
```

10.6   EWA

⟨sf_ewa.f90⟩≡
  ⟨*File header*⟩

  module sf_ewa

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants, only: pi !NODEP!
  ⟨*Use file utils*⟩
    use limits, only: FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use pdg_arrays
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base

  ⟨*Standard module head*⟩

  ⟨*SF ewa: public*⟩

  ⟨*SF ewa: parameters*⟩

  ⟨*SF ewa: types*⟩

  contains

  ⟨*SF ewa: procedures*⟩

  ⟨*SF ewa: tests*⟩

  end module sf_ewa

10.6.1   Physics

The EWA structure function for a Z or W inside a fermion (lepton or quark)
depends on the vector-boson polarization. We distinguish transversal ($\pm$) and

longitudinal (0) polarization.

$$F_+(x) = \frac{1}{16\pi^2} \frac{(v-a)^2 + (v+a)^2 \bar{x}^2}{x} \left[\ln\left(\frac{p_{\perp,\max}^2 + \bar{x}M^2}{\bar{x}M^2} \right) - \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x}M^2} \right]$$

(10.62)

$$F_-(x) = \frac{1}{16\pi^2} \frac{(v+a)^2 + (v-a)^2 \bar{x}^2}{x} \left[\ln\left(\frac{p_{\perp,\max}^2 + \bar{x}M^2}{\bar{x}M^2} \right) - \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x}M^2} \right]$$

(10.63)

$$F_0(x) = \frac{v^2 + a^2}{8\pi^2} \frac{2\bar{x}}{x} \frac{p_{\perp,\max}^2}{p_{\perp,\max}^2 + \bar{x}M^2}$$

(10.64)

where $p_{\perp,\max}$ is the cutoff in transversal momentum, M is the vector-boson mass, v and a are the vector and axial-vector couplings, and $\bar{x} \equiv 1-x$. Note that the longitudinal structure function is finite for large cutoff, while the transversal structure function is logarithmically divergent.

The maximal transverse momentum is given by the kinematical limit, it is

$$p_{\perp,\max} = \bar{x}\sqrt{s}/2.$$

(10.65)

The vector and axial couplings for a fermion branching into a W are

$$v_W = \frac{g}{2\sqrt{2}}, \qquad\qquad a_W = \frac{g}{2\sqrt{2}}.$$

(10.66)

For Z emission, this is replaced by

$$v_Z = \frac{g}{2\cos\theta_w} \left(t_3 - 2q\sin^2\theta_w \right), \qquad a_Z = \frac{g}{2\cos\theta_w} t_3,$$

(10.67)

where $t_3 = \pm\frac{1}{2}$ is the fermion isospin, and q its charge.

For an initial antifermion, the signs of the axial couplings are inverted. Note that a common sign change of v and a is irrelevant.

The EWA depends on the parameters g, $\sin^2\theta_w$, M_W, and M_Z. These can all be taken from the SM input, and the prefactors are calculated from those and the incoming particle type.

Since these structure functions have a $1/x$ singularity (which is not really relevant in practice, however, since the vector boson mass is finite), we map this singularity allowing for nontrivial x bounds:

$$x = \exp(\bar{r}\ln x_0 + r\ln x_1)$$

(10.68)

such that

$$\int_{x_0}^{x_1} \frac{dx}{x} = (\ln x_1 - \ln x_0) \int_0^1 dr.$$

(10.69)

As a user parameter, we have the cutoff $p_{\perp,\max}$. The divergence $1/x$ also requires a x_0 cutoff; and for completeness we introduce a corresponding x_1. Physically, the minimal sensible value of x is M^2/s, although the approximation loses its value already at higher x values.

10.6.2 The EWA data block

The EWA parameters are: $p_{T,\max}$, c_V, c_A, and m. Instead of m we can use the incoming particle PDG code as input; from this we can deduce the mass and charges. In the initialization phase it is not yet determined whether a W or a Z is radiated, hence we set the vector and axial-vector couplings equal to the common prefactors $g/2 = e/2/\sin\theta_W$.

In principle, for EWA it would make sense to allow the user to also set the upper bound for x, $x_{\max}$, but we fix it to one here.

⟨*SF ewa: public*⟩≡
```
    public :: ewa_data_t
```

⟨*SF ewa: types*⟩≡
```
    type, extends(sf_data_t) :: ewa_data_t
      private
      type(model_t), pointer :: model => null ()
      type(flavor_t), dimension(:), allocatable :: flv_in
      type(flavor_t), dimension(:), allocatable :: flv_out
      real(default) :: pt_max
      real(default) :: sqrts
      real(default) :: x_min
      real(default) :: x_max
      real(default) :: mass
      real(default) :: m_out
      real(default) :: q_min
      real(default) :: cv
      real(default) :: ca
      real(default) :: costhw
      real(default) :: sinthw
      real(default) :: mW
      real(default) :: mZ
      real(default) :: coeff
      logical :: mass_set = .false.
      logical :: keep_momentum
      logical :: keep_energy
      integer :: id = 0
      integer :: error = NONE
    contains
    ⟨SF ewa: ewa data: TBP⟩
    end type ewa_data_t
```

Error codes

⟨*SF ewa: parameters*⟩≡
```
    integer, parameter :: NONE = 0
    integer, parameter :: ZERO_QMIN = 1
    integer, parameter :: Q_MAX_TOO_SMALL = 2
    integer, parameter :: ZERO_XMIN = 3
    integer, parameter :: MASS_MIX = 4
    integer, parameter :: ZERO_SW = 5
    integer, parameter :: ISOSPIN_MIX = 6
    integer, parameter :: WRONG_PRT = 7
    integer, parameter :: MASS_MIX_OUT = 8
    integer, parameter :: NO_EWA = 9
```

⟨*SF ewa: ewa data: TBP*⟩≡
```
    procedure :: init => ewa_data_init
```

⟨*SF ewa: procedures*⟩≡
```
    subroutine ewa_data_init (data, model, pdg_in, x_min, pt_max, &
         sqrts, keep_momentum, keep_energy, mass)
      class(ewa_data_t), intent(inout) :: data
      type(model_t), intent(in), target :: model
      type(pdg_array_t), intent(in) :: pdg_in
      real(default), intent(in) :: x_min, pt_max, sqrts
      logical, intent(in) :: keep_momentum, keep_energy
      real(default), intent(in), optional :: mass
      real(default) :: g, ee
      integer :: n_flv, i
      data%model => model
      if (.not. any (pdg_in .match. &
           [1,2,3,4,5,6,11,13,15,-1,-2,-3,-4,-5,-6,-11,-13,-15])) then
         data%error = WRONG_PRT;  return
      end if
      n_flv = pdg_array_get_length (pdg_in)
      allocate (data%flv_in (n_flv))
      allocate (data%flv_out(n_flv))
      do i = 1, n_flv
         call flavor_init (data%flv_in(i), pdg_array_get (pdg_in, i), model)
      end do
      data%pt_max = pt_max
      data%sqrts = sqrts
      data%x_min = x_min
      data%x_max = 1
      if (data%x_min == 0) then
         data%error = ZERO_XMIN;  return
      end if
      select case (char (data%model%get_name ()))
      case ("QCD","QED","Test")
         data%error = NO_EWA;  return
      end select
      ee = model_get_parameter_value (data%model, var_str ("ee"))
      data%sinthw = model_get_parameter_value (data%model, var_str ("sw"))
      data%costhw = model_get_parameter_value (data%model, var_str ("cw"))
      data%mZ = model_get_parameter_value (data%model, var_str ("mZ"))
      data%mW = model_get_parameter_value (data%model, var_str ("mW"))
      if (data%sinthw /= 0) then
         g = ee / data%sinthw
      else
         data%error = ZERO_SW;  return
      end if
      data%cv = g / 2._default
      data%ca = g / 2._default
      data%coeff = 1._default / (8._default * PI**2)
      data%keep_momentum = keep_momentum
      data%keep_energy = keep_energy
      if (present (mass)) then
         data%mass = mass
         data%m_out = mass
         data%mass_set = .true.
```

```
      else
         data%mass = flavor_get_mass (data%flv_in(1))
         if (any (flavor_get_mass (data%flv_in) /= data%mass)) then
            data%error = MASS_MIX;  return
         end if
      end if
   end subroutine ewa_data_init
```

Set the vector boson ID for distinguishing W and Z bosons.

⟨*SF ewa: ewa data: TBP*⟩+≡
```
   procedure :: set_id => ewa_set_id
```

⟨*SF ewa: procedures*⟩+≡
```
   subroutine ewa_set_id (data, id)
     class(ewa_data_t), intent(inout) :: data
     integer, intent(in) :: id
     integer :: i, isospin, pdg
     if (.not. allocated (data%flv_in)) &
          call msg_bug ("EWA: incoming particles not set")
     data%id = id
     select case (data%id)
     case (23)
        data%m_out = data%mass
        data%flv_out = data%flv_in
     case (24)
        do i = 1, size (data%flv_in)
           pdg = flavor_get_pdg (data%flv_in(i))
           isospin = flavor_get_isospin_type (data%flv_in(i))
           if (isospin > 0) then
              !!! up-type quark or neutrinos
              if (flavor_is_antiparticle (data%flv_in(i))) then
                 call flavor_init (data%flv_out(i), pdg + 1, data%model)
              else
                 call flavor_init (data%flv_out(i), pdg - 1, data%model)
              end if
           else
              !!! down-type quark or lepton
              if (flavor_is_antiparticle (data%flv_in(i))) then
                 call flavor_init (data%flv_out(i), pdg - 1, data%model)
              else
                 call flavor_init (data%flv_out(i), pdg + 1, data%model)
              end if
           end if
        end do
        if (.not. data%mass_set) then
           data%m_out = flavor_get_mass (data%flv_out(1))
           if (any (flavor_get_mass (data%flv_out) /= data%m_out)) then
              data%error = MASS_MIX_OUT;  return
           end if
        end if
     end select
   end subroutine ewa_set_id
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*SF ewa: ewa data: TBP*⟩+≡
```
procedure :: check => ewa_data_check
```

⟨*SF ewa: procedures*⟩+≡
```
subroutine ewa_data_check (data)
  class(ewa_data_t), intent(in) :: data
  select case (data%error)
  case (WRONG_PRT)
     call msg_fatal ("EWA structure function only accessible for " &
          // "SM quarks and leptons.")
  case (NO_EWA)
     call msg_fatal ("EWA structure function not available for model " &
          // char (data%model%get_name ()))
  case (ZERO_SW)
     call msg_fatal ("EWA: Vanishing value of sin(theta_w)")
  case (ZERO_QMIN)
     call msg_fatal ("EWA: Particle mass is zero")
  case (Q_MAX_TOO_SMALL)
     call msg_fatal ("EWA: Particle mass exceeds Qmax")
  case (ZERO_XMIN)
     call msg_fatal ("EWA: x_min must be larger than zero")
  case (MASS_MIX)
     call msg_fatal ("EWA: incoming particle masses must be uniform")
  case (MASS_MIX_OUT)
     call msg_fatal ("EWA: outgoing particle masses must be uniform")
  case (ISOSPIN_MIX)
     call msg_fatal ("EWA: incoming particle isospins must be uniform")
  end select
end subroutine ewa_data_check
```

Output

⟨*SF ewa: ewa data: TBP*⟩+≡
```
procedure :: write => ewa_data_write
```

⟨*SF ewa: procedures*⟩+≡
```
subroutine ewa_data_write (data, unit, verbose)
  class(ewa_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "EWA data:"
  if (allocated (data%flv_in) .and. allocated (data%flv_out)) then
     write (u, "(3x,A)", advance="no") "  flavor(in)  =  "
     do i = 1, size (data%flv_in)
        if (i > 1)  write (u, "(',',1x)", advance="no")
        call flavor_write (data%flv_in(i), u)
     end do
     write (u, *)
     write (u, "(3x,A)", advance="no") "  flavor(out) =  "
     do i = 1, size (data%flv_out)
        if (i > 1)  write (u, "(',',1x)", advance="no")
```

```
          call flavor_write (data%flv_out(i), u)
        end do
        write (u, *)
        write (u, "(3x,A," // FMT_19 // ")") "  x_min     = ", data%x_min
        write (u, "(3x,A," // FMT_19 // ")") "  x_max     = ", data%x_max
        write (u, "(3x,A," // FMT_19 // ")") "  pt_max    = ", data%pt_max
        write (u, "(3x,A," // FMT_19 // ")") "  sqrts     = ", data%sqrts
        write (u, "(3x,A," // FMT_19 // ")") "  mass      = ", data%mass
        write (u, "(3x,A," // FMT_19 // ")") "  cv        = ", data%cv
        write (u, "(3x,A," // FMT_19 // ")") "  ca        = ", data%ca
        write (u, "(3x,A," // FMT_19 // ")") "  coeff     = ", data%coeff
        write (u, "(3x,A," // FMT_19 // ")") "  costhw    = ", data%costhw
        write (u, "(3x,A," // FMT_19 // ")") "  sinthw    = ", data%sinthw
        write (u, "(3x,A," // FMT_19 // ")") "  mZ        = ", data%mZ
        write (u, "(3x,A," // FMT_19 // ")") "  mW        = ", data%mW
        write (u, "(3x,A,L2)")        "  keep_mom. = ", data%keep_momentum
        write (u, "(3x,A,L2)")        "  keep_en.  = ", data%keep_energy
        write (u, "(3x,A,I2)")        "  PDG (VB)  = ", data%id
      else
        write (u, "(3x,A)") "[undefined]"
      end if
    end subroutine ewa_data_write
```

The number of parameters is one for collinear splitting, in case one of the options
`keep_energy` or `keep_momentum` is set, we take the recoil into account.

⟨*SF ewa: ewa data: TBP*⟩+≡
```
    procedure :: get_n_par => ewa_data_get_n_par
```

⟨*SF ewa: procedures*⟩+≡
```
    function ewa_data_get_n_par (data) result (n)
      class(ewa_data_t), intent(in) :: data
      integer :: n
      if (data%keep_energy .or. data%keep_momentum) then
        n = 3
      else
        n = 1
      end if
    end function ewa_data_get_n_par
```

Return the outgoing particles PDG codes. This depends, whether this is a
charged-current or neutral-current interaction.

⟨*SF ewa: ewa data: TBP*⟩+≡
```
    procedure :: get_pdg_out => ewa_data_get_pdg_out
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_data_get_pdg_out (data, pdg_out)
      class(ewa_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      integer, dimension(:), allocatable :: pdg1
      integer :: i, n_flv
      if (allocated (data%flv_out)) then
        n_flv = size (data%flv_out)
      else
        n_flv = 0
```

```
      end if
      allocate (pdg1 (n_flv))
      do i = 1, n_flv
         pdg1(i) = flavor_get_pdg (data%flv_out(i))
      end do
      pdg_out(1) = pdg1
    end subroutine ewa_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF ewa: ewa data: TBP*⟩+≡
```
    procedure :: allocate_sf_int => ewa_data_allocate_sf_int
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_data_allocate_sf_int (data, sf_int)
      class(ewa_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
      allocate (ewa_t :: sf_int)
    end subroutine ewa_data_allocate_sf_int
```

10.6.3   The EWA object

The `ewa_t` data type is a $1 \to 2$ interaction. We should be able to handle several flavors in parallel, since EWA is not necessarily applied immediately after beam collision: W/Z bosons may be radiated from quarks. In that case, the partons are massless and $q_{\min}$ applies instead, so we do not need to generate several kinematical configurations in parallel.

The particles are ordered as (incoming, radiated, W/Z), where the W/Z initiates the hard interaction.

In the case of EPA, we generated an unpolarized photon and transferred initial polarization to the radiated parton. Color is transferred in the same way. I do not know whether the same can/should be done for EWA, as the structure functions depend on the W/Z polarization. If we are having Z bosons, both up- and down-type fermions can participate. Otherwise, with a W^+ an up-type fermion is transferred to a down-type fermion, and the other way round.

⟨*SF ewa: types*⟩+≡
```
    type, extends (sf_int_t) :: ewa_t
      type(ewa_data_t), pointer :: data => null ()
      real(default) :: x  = 0
      real(default) :: xb = 0
      integer :: n_me = 0
      real(default), dimension(:), allocatable :: cv
      real(default), dimension(:), allocatable :: ca
    contains
    ⟨SF ewa: ewa: TBP⟩
    end type ewa_t
```

Type string: has to be here, but there is no string variable on which EWA depends. Hence, a dummy routine.

⟨*SF ewa: ewa: TBP*⟩≡
```
    procedure :: type_string => ewa_type_string
```

⟨*SF ewa: procedures*⟩+≡
```
    function ewa_type_string (object) result (string)
      class(ewa_t), intent(in) :: object
      type(string_t) :: string
      if (associated (object%data)) then
         string = "EWA: equivalent W/Z approx."
      else
         string = "EWA: [undefined]"
      end if
    end function ewa_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF ewa: ewa: TBP*⟩+≡
```
    procedure :: write => ewa_write
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_write (object, unit, testflag)
      class(ewa_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      if (associated (object%data)) then
         call object%data%write (u)
         if (object%status >= SF_DONE_KINEMATICS) then
            write (u, "(1x,A)")  "SF parameters:"
            write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
         end if
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "EWA data: [undefined]"
      end if
    end subroutine ewa_write
```

The current implementation requires uniform isospin for all incoming particles, therefore we need to probe only the first one.

⟨*SF ewa: ewa: TBP*⟩+≡
```
    procedure :: init => ewa_init
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_init (sf_int, data)
      class(ewa_t), intent(out) :: sf_int
      class(sf_data_t), intent(in), target :: data
      type(quantum_numbers_mask_t), dimension(3) :: mask
      integer, dimension(3) :: hel_lock
      type(polarization_t) :: pol
      type(quantum_numbers_t), dimension(1) :: qn_fc, qn_hel, qn_fc_fin
      type(flavor_t) :: flv_z, flv_wp, flv_wm
      type(quantum_numbers_t) :: qn_z, qn_wp, qn_wm, qn, qn_out, qn_w
      type(state_iterator_t) :: it_hel
      integer :: i, isospin
      select type (data)
      type is (ewa_data_t)
         mask = new_quantum_numbers_mask (.false., .false., &
```

```
      mask_h = [.false., .false., .true.])
hel_lock = [2, 1, 0]
select case (data%id)
case (23)
   !!! Z boson, flavor is not changing
   call sf_int%base_init (mask, [data%mass**2], [data%mass**2], &
       [data%mZ**2], hel_lock = hel_lock)
   sf_int%data => data
   call flavor_init (flv_z, Z_BOSON, data%model)
   call quantum_numbers_init (qn_z, flv_z)
   do i = 1, size (data%flv_in)
      call polarization_init_generic (pol, data%flv_in(i))
      call quantum_numbers_init (qn_fc(1), &
          flv = data%flv_in(i), &
          col = color_from_flavor (data%flv_in(i), 1))
      call state_iterator_init (it_hel, pol%state)
      do while (state_iterator_is_valid (it_hel))
         qn_hel = state_iterator_get_quantum_numbers (it_hel)
         qn = qn_hel(1) .merge. qn_fc(1)
         call interaction_add_state &
             (sf_int%interaction_t, [qn, qn, qn_z])
         call state_iterator_advance (it_hel)
      end do
      call polarization_final (pol)
   end do
case (24)
   call sf_int%base_init (mask, [data%mass**2], [data%m_out**2], &
       [data%mW**2], hel_lock = hel_lock)
   sf_int%data => data
   call flavor_init (flv_wp, W_BOSON, data%model)
   call flavor_init (flv_wm, - W_BOSON, data%model)
   call quantum_numbers_init (qn_wp, flv_wp)
   call quantum_numbers_init (qn_wm, flv_wm)
   do i = 1, size (data%flv_in)
      isospin = flavor_get_isospin_type (data%flv_in(i))
      if (isospin > 0) then
         !!! up-type quark or neutrinos
         if (flavor_is_antiparticle (data%flv_in(i))) then
            qn_w = qn_wm
         else
            qn_w = qn_wp
         end if
      else
         !!! down-type quark or lepton
         if (flavor_is_antiparticle (data%flv_in(i))) then
            qn_w = qn_wp
         else
            qn_w = qn_wm
         end if
      end if
      call polarization_init_generic (pol, data%flv_in(i))
      call quantum_numbers_init (qn_fc(1), &
          flv = data%flv_in(i), &
          col = color_from_flavor (data%flv_in(i), 1))
```

1231

```
            call quantum_numbers_init (qn_fc_fin(1), &
                 flv = data%flv_out(i), &
                 col = color_from_flavor (data%flv_out(i), 1))
            call state_iterator_init (it_hel, pol%state)
            do while (state_iterator_is_valid (it_hel))
               qn_hel = state_iterator_get_quantum_numbers (it_hel)
               qn = qn_hel(1) .merge. qn_fc(1)
               qn_out = qn_hel(1) .merge. qn_fc_fin(1)
               call interaction_add_state &
                    (sf_int%interaction_t, [qn, qn_out, qn_w])
               call state_iterator_advance (it_hel)
            end do
            call polarization_final (pol)
         end do
      case default
         call msg_fatal ("EWA initialization failed: wrong particle type.")
      end select
      call interaction_freeze (sf_int%interaction_t)
      if (data%keep_momentum) then
         if (data%keep_energy) then
            call msg_fatal ("EWA: momentum and energy" // &
                 "cannot be simultaneously conserved.")
         else
            sf_int%on_shell_mode = KEEP_MOMENTUM
         end if
      else
         if (data%keep_energy) then
            sf_int%on_shell_mode = KEEP_ENERGY
         end if
      end if
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
    end select
  end subroutine ewa_init
```

Prepare the coupling arrays. This is separate from the previous routine since the state matrix may be helicity-contracted.

⟨*SF ewa: ewa: TBP*⟩+≡
```
    procedure :: setup_constants => ewa_setup_constants
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_setup_constants (sf_int)
      class(ewa_t), intent(inout) :: sf_int
      type(state_iterator_t) :: it
      type(flavor_t) :: flv
      real(default) :: q, t3
      integer :: i
      sf_int%n_me = interaction_get_n_matrix_elements (sf_int%interaction_t)
      allocate (sf_int%cv (sf_int%n_me))
      allocate (sf_int%ca (sf_int%n_me))
      associate (data => sf_int%data)
        select case (data%id)
        case (23)
```

```
             call state_iterator_init (it, &
                 interaction_get_state_matrix_ptr (sf_int%interaction_t))
            do while (state_iterator_is_valid (it))
               i = state_iterator_get_me_index (it)
               flv = state_iterator_get_flavor (it, 1)
               q = flavor_get_charge (flv)
               t3 = flavor_get_isospin (flv)
               if (flavor_is_antiparticle (flv)) then
                  sf_int%cv(i) = - data%cv &
                       * (t3 - 2._default * q * data%sinthw**2) / data%costhw
                  sf_int%ca(i) = data%ca *  t3 / data%costhw
               else
                  sf_int%cv(i) = data%cv &
                       * (t3 - 2._default * q * data%sinthw**2) / data%costhw
                  sf_int%ca(i) = data%ca *  t3 / data%costhw
               end if
               call state_iterator_advance (it)
            end do
         case (24)
            call state_iterator_init (it, &
                 interaction_get_state_matrix_ptr (sf_int%interaction_t))
            do while (state_iterator_is_valid (it))
               i = state_iterator_get_me_index (it)
               if (flavor_is_antiparticle (state_iterator_get_flavor (it, 1))) &
                    then
                  sf_int%cv(i) = data%cv / sqrt(2._default)
                  sf_int%ca(i) = - data%ca / sqrt(2._default)
               else
                  sf_int%cv(i) = data%cv / sqrt(2._default)
                  sf_int%ca(i) = data%ca / sqrt(2._default)
               end if
               call state_iterator_advance (it)
            end do
         end select
      end associate
      sf_int%status = SF_INITIAL
   end subroutine ewa_setup_constants
```

10.6.4 Kinematics

Set kinematics. The EWA structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the x value is transformed. Higher orders affect the function value. If map is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If map is set, the exponential mapping for the $1/x$ singularity discussed above is applied.

⟨*SF ewa: ewa: TBP*⟩+≡
```
    procedure :: complete_kinematics => ewa_complete_kinematics
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_complete_kinematics (sf_int, x, f, r, rb, map)
      class(ewa_t), intent(inout) :: sf_int
```

```fortran
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    real(default), dimension(:), intent(in) :: rb
    logical, intent(in) :: map
    real(default) :: xb1, e_1
    real(default) :: x0, x1, lx0, lx1, lx
    e_1 = energy (interaction_get_momentum (sf_int%interaction_t, 1))
    if (sf_int%data%keep_momentum .or. sf_int%data%keep_energy) then
       select case (sf_int%data%id)
       case (23)
          x0 = max (sf_int%data%x_min, sf_int%data%mz / e_1)
       case (24)
          x0 = max (sf_int%data%x_min, sf_int%data%mw / e_1)
       end select
    else
       x0 = sf_int%data%x_min
    end if
    x1 = sf_int%data%x_max
    if ( x0 >= x1) then
       f = 0
       sf_int%status = SF_FAILED_KINEMATICS
       return
    end if
    if (map) then
       lx0 = log (x0)
       lx1 = log (x1)
       lx = lx1 * r(1) + lx0 * rb(1)
       x(1) = exp(lx)
       f = x(1) * (lx1 - lx0)
    else
       x(1) = r(1)
       if (x0 < x(1) .and. x(1) < x1) then
          f = 1
       else
          sf_int%status = SF_FAILED_KINEMATICS
          f = 0
          return
       end if
    end if
    xb1 = 1 - x(1)
    if (size(x) == 3)  x(2:3) = r(2:3)
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
       sf_int%x  = x(1)
       sf_int%xb = xb1
    case (SF_FAILED_KINEMATICS)
       sf_int%x  = 0
       sf_int%xb = 0
       f = 0
    end select
  end subroutine ewa_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF ewa: ewa: TBP*⟩+≡
```
    procedure :: inverse_kinematics => ewa_inverse_kinematics
```

⟨*SF ewa: procedures*⟩+≡
```
    subroutine ewa_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
      class(ewa_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      real(default) :: x0, x1, lx0, lx1, lx, e_1
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      e_1 = energy (interaction_get_momentum (sf_int%interaction_t, 1))
      if (sf_int%data%keep_momentum .or. sf_int%data%keep_energy) then
         select case (sf_int%data%id)
         case (23)
            x0 = max (sf_int%data%x_min, sf_int%data%mz / e_1)
         case (24)
            x0 = max (sf_int%data%x_min, sf_int%data%mw / e_1)
         end select
      else
         x0 = sf_int%data%x_min
      end if
      x1 = sf_int%data%x_max
      if (map) then
         lx0 = log (x0)
         lx1 = log (x1)
         lx = log (x(1))
         r(1)  = (lx - lx0) / (lx1 - lx0)
         rb(1) = (lx1 - lx) / (lx1 - lx0)
         f = x(1) * (lx1 - lx0)
      else
         r (1) = x(1)
         rb(1) = 1 - x(1)
         if (x0 < x(1) .and. x(1) < x1) then
            f = 1
         else
            f = 0
         end if
      end if
      if (size(r) == 3) then
         r (2:3) = x(2:3)
         rb(2:3) = 1 - x(2:3)
      end if
      if (set_mom) then
         call sf_int%split_momentum (x, sf_int%xb)
         select case (sf_int%status)
         case (SF_DONE_KINEMATICS)
```

```
            sf_int%x  = x(1)
            sf_int%xb = 1 - x(1)
         case (SF_FAILED_KINEMATICS)
            sf_int%x = 0
            f = 0
         end select
      end if
   end subroutine ewa_inverse_kinematics
```

10.6.5 EWA application

For EWA, we can compute kinematics and function value in a single step. This function works on a single beam, assuming that the input momentum has been set. We need four random numbers as input: one for x, one for Q^2, and two for the polar and azimuthal angles. Alternatively, we can skip p_T generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated x values are identical.

⟨*SF ewa: ewa: TBP*⟩+≡
```
   procedure :: apply => ewa_apply
```

⟨*SF ewa: procedures*⟩+≡
```
   subroutine ewa_apply (sf_int, scale)
     class(ewa_t), intent(inout) :: sf_int
     real(default), intent(in) :: scale
     real(default) :: x, xb, pt2, c1, c2
     real(default) :: cv, ca
     real(default) :: f, fm, fp, fL
     integer :: i
     associate (data => sf_int%data)
       x  = sf_int%x
       xb = sf_int%xb
       pt2 = min ((data%pt_max)**2, (xb * data%sqrts / 2)**2)
       select case (data%id)
       case (23)
          !!! Z boson structure function
          c1 = log (1 + pt2 / (xb * (data%mZ)**2))
          c2 = 1 / (1 + (xb * (data%mZ)**2) / pt2)
       case (24)
          !!! W boson structure function
          c1 = log (1 + pt2 / (xb * (data%mW)**2))
          c2 = 1 / (1 + (xb * (data%mW)**2) / pt2)
       end select
       do i = 1, sf_int%n_me
          cv = sf_int%cv(i)
          ca = sf_int%ca(i)
          fm = data%coeff * &
               ((cv + ca)**2 + ((cv - ca) * xb)**2) * (c1 - c2) / (2 * x)
          fp = data%coeff * &
               ((cv - ca)**2 + ((cv + ca) * xb)**2) * (c1 - c2) / (2 * x)
          fL = data%coeff * &
```

```
                (cv**2 + ca**2) * (2 * xb / x) * c2
          f = fp + fm + fL
          if (f /= 0) then
             fp = fp / f
             fm = fm / f
             fL = fL / f
          end if
          call interaction_set_matrix_element &
               (sf_int%interaction_t, i, cmplx (f, kind=default))
       end do
    end associate
    sf_int%status = SF_EVALUATED
  end subroutine ewa_apply
```

10.6.6   Unit tests

⟨*SF ewa: public*⟩+≡
```
    public :: sf_ewa_test
```

⟨*SF ewa: tests*⟩≡
```
    subroutine sf_ewa_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF ewa: execute tests⟩
    end subroutine sf_ewa_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF ewa: execute tests*⟩≡
```
    call test (sf_ewa_1, "sf_ewa_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF ewa: tests*⟩+≡
```
    subroutine sf_ewa_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      type(pdg_array_t), dimension(1) :: pdg_out
      integer, dimension(:), allocatable :: pdg1
      class(sf_data_t), allocatable :: data

      write (u, "(A)")  "* Test output: sf_ewa_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
          &test structure function data"
      write (u, "(A)")

      write (u, "(A)")  "* Create empty data object"
      write (u, "(A)")
```

```
call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("SM"), &
     var_str ("SM.mdl"), os_data, model)
pdg_in = 2

allocate (ewa_data_t :: data)
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize for Z boson"
write (u, "(A)")

select type (data)
type is (ewa_data_t)
   call data%init (model, pdg_in, 0.01_default, &
        500._default, 5000._default, .false., .false.)
   call data%set_id (23)
end select

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)")  "Outgoing particle codes:"
call data%get_pdg_out (pdg_out)
pdg1 = pdg_out(1)
write (u, "(2x,99(1x,I0))")  pdg1

write (u, "(A)")
write (u, "(A)")  "* Initialize for W boson"
write (u, "(A)")

deallocate (data)
allocate (ewa_data_t :: data)
select type (data)
type is (ewa_data_t)
   call data%init (model, pdg_in, 0.01_default, &
        500._default, 5000._default, .false., .false.)
   call data%set_id (24)
end select

call data%write (u)

write (u, "(A)")

write (u, "(1x,A)")  "Outgoing particle codes:"
call data%get_pdg_out (pdg_out)
pdg1 = pdg_out(1)
write (u, "(2x,99(1x,I0))")  pdg1

call model_list%final ()
call syntax_model_file_final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_ewa_1"

  end subroutine sf_ewa_1
```

Test and probe structure function

Construct and display a structure function object based on the EWA structure
function.

⟨*SF ewa: execute tests*⟩+≡
```
    call test (sf_ewa_2, "sf_ewa_2", &
        "structure function instance", &
        u, results)
```

⟨*SF ewa: tests*⟩+≡
```
  subroutine sf_ewa_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_ewa_2"
    write (u, "(A)")  "*   Purpose: initialize and fill &
        &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list%read_model (var_str ("SM"), &
        var_str ("SM.mdl"), os_data, model)
    call flavor_init (flv, 2, model)
    pdg_in = 2

    call reset_interaction_counter ()

    allocate (ewa_data_t :: data)
    select type (data)
    type is (ewa_data_t)
       call data%init (model, pdg_in, 0.01_default, &
           500._default, 3000._default, .false., .false.)
```

```
      call data%set_id (24)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=1500"
write (u, "(A)")
E = 1500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, no EWA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
```

```
      call sf_int%recover_x (x)
      call sf_int%inverse_kinematics (x, f, r, rb, map=.false., &
           set_momenta=.true.)

      write (u, "(A,9(1x,F10.7))")  "r =", r
      write (u, "(A,9(1x,F10.7))")  "rb=", rb
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate EWA structure function"
      write (u, "(A)")

      call sf_int%apply (scale = 100._default)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_ewa_2"

    end subroutine sf_ewa_2
```

Standard mapping

Construct and display a structure function object based on the EWA structure
function, applying the standard single-particle mapping.

⟨*SF ewa: execute tests*⟩+≡
```
    call test (sf_ewa_3, "sf_ewa_3", &
         "apply mapping", &
         u, results)
```

⟨*SF ewa: tests*⟩+≡
```
    subroutine sf_ewa_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f
```

```
write (u, "(A)")  "* Test output: sf_ewa_3"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &test structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("SM"), &
     var_str ("SM.mdl"), os_data, model)
call flavor_init (flv, 2, model)
pdg_in = 2

call reset_interaction_counter ()

allocate (ewa_data_t :: data)
select type (data)
type is (ewa_data_t)
   call data%init (model, pdg_in, 0.01_default, &
        500._default, 3000._default, .false., .false.)
   call data%set_id (24)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=1500"
write (u, "(A)")
E = 1500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, with EWA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
```

```
      call sf_int%complete_kinematics (x, f, r, rb, map=.true.)

    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "rb=", rb
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Recover x from momenta"
    write (u, "(A)")

    q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
    call sf_int%final ()
    deallocate (sf_int)

    call data%allocate_sf_int (sf_int)
    call sf_int%init (data)
    call sf_int%set_beam_index ([1])
    call sf_int%setup_constants ()

    call sf_int%seed_kinematics ([k])
    call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
    call sf_int%recover_x (x)
    call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
         set_momenta=.true.)

    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "rb=", rb
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate EWA structure function"
    write (u, "(A)")

    call sf_int%apply (scale = 100._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_ewa_3"

  end subroutine sf_ewa_3
```

Non-collinear case

Construct and display a structure function object based on the EPA structure function.

⟨*SF ewa: execute tests*⟩+≡
```
    call test (sf_ewa_4, "sf_ewa_4", &
        "non-collinear", &
        u, results)
```

⟨*SF ewa: tests*⟩+≡
```
    subroutine sf_ewa_4 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_ewa_4"
      write (u, "(A)")  "*   Purpose: initialize and fill &
          &test structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("SM"), &
          var_str ("SM.mdl"), os_data, model)
      call flavor_init (flv, 2, model)
      pdg_in = 2

      call reset_interaction_counter ()

      allocate (ewa_data_t :: data)
      select type (data)
      type is (ewa_data_t)
         call data%init (model, pdg_in, 0.01_default, &
             500._default, 3000.0_default, .false., .true.)
         call data%set_id (24)
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
```

```
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=1500"
write (u, "(A)")
E = 1500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.5/0.5/0.25, with EWA mapping, "
write (u, "(A)")  "           non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
     set_momenta=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
```

```
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate EWA structure function"
      write (u, "(A)")

      call sf_int%apply (scale = 1500._default)
      call sf_int%write (u, testflag = .true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_ewa_4"

    end subroutine sf_ewa_4
```

Structure function for multiple flavors

Construct and display a structure function object based on the EWA structure function. The incoming state has multiple particles with non-uniform quantum numbers.

⟨*SF ewa: execute tests*⟩+≡
```
    call test (sf_ewa_5, "sf_ewa_5", &
         "structure function instance", &
         u, results)
```

⟨*SF ewa: tests*⟩+≡
```
    subroutine sf_ewa_5 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_ewa_5"
      write (u, "(A)")  "*   Purpose: initialize and fill &
           &test structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")
```

```
call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("SM"), &
     var_str ("SM.mdl"), os_data, model)
call flavor_init (flv, 2, model)
pdg_in = [1, 2, -1, -2]

call reset_interaction_counter ()

allocate (ewa_data_t :: data)
select type (data)
type is (ewa_data_t)
   call data%init (model, pdg_in, 0.01_default, &
        500._default, 3000._default, .false., .false.)
   call data%set_id (24)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=1500"
write (u, "(A)")
E = 1500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, no EWA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Evaluate EWA structure function"
  write (u, "(A)")

  call sf_int%apply (scale = 100._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_ewa_5"

end subroutine sf_ewa_5
```

10.7 Lepton collider beamstrahlung: CIRCE1

⟨sf_circe1.f90⟩≡
  ⟨*File header*⟩

  module sf_circe1

  ⟨*Use kinds*⟩
    use kinds, only: double !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use os_interface
    use unit_tests
    use lorentz !NODEP!
    use rng_base
    use pdg_arrays
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_mappings
    use sf_aux
    use sf_base
    use circe1, circe1_rng_t => rng_type !NODEP!

  ⟨*Standard module head*⟩

  ⟨*SF circe1: public*⟩

  ⟨*SF circe1: types*⟩

  contains

  ⟨*SF circe1: procedures*⟩

  ⟨*SF circe1: tests*⟩

  end module sf_circe1

10.7.1 Physics

Beamstrahlung is applied before ISR. The CIRCE1 implementation has a single structure function for both beams (which makes sense since it has to be switched on or off for both beams simultaneously). Nevertheless it is factorized:

The functional form in the CIRCE1 parameterization is defined for electrons or photons

$$f(x) = \alpha \, x^{\beta} \, (1 - x)^{\gamma} \tag{10.70}$$

for $x < 1 - \epsilon$ (resp. $x > \epsilon$ in the photon case). In the remaining interval, the standard form is zero, with a delta singularity at $x = 1$ (resp. $x = 0$). Equivalently, the delta part may be distributed uniformly among this interval. This latter form is implemented in the `kirke` version of the `CIRCE1` subroutines, and is used here.

The parameter `circe1\_eps` sets the peak mapping of the `CIRCE1` structure function. Its default value is 10^{-5}. The other parameters are the parameterization version and revision number, the accelerator type, and the $\sqrt{s}$ value used by `CIRCE1`. The chattiness can also be set.

Since the energy is distributed in a narrow region around unity (for electrons) or zero (for photons), it is advantageous to map the interval first. The mapping is controlled by the parameter `circe1\_epsilon` which is taken from the `CIRCE1` internal data structure.

The $\sqrt{s}$ value, if not explicitly set, is taken from the process data. Note that interpolating $\sqrt{s}$ is not recommended; one should rather choose one of the distinct values known to `CIRCE1`.

10.7.2 The CIRCE1 data block

The CIRCE1 parameters are: The incoming flavors, the flags whether the photon or the lepton is the parton in the hard interaction, the flags for the generation mode (generator/mapping/no mapping), the mapping parameter ϵ, $\sqrt{s}$ and several steering parameters: `ver`, `rev`, `acc`, `chat`.

In generator mode, the x values are actually discarded and a random number generator is used instead.

⟨*SF circe1: public*⟩≡
```
    public :: circe1_data_t
```

⟨*SF circe1: types*⟩≡
```
    type, extends (sf_data_t) :: circe1_data_t
       private
       type(model_t), pointer :: model => null ()
       type(flavor_t), dimension(2) :: flv_in
       integer, dimension(2) :: pdg_in
       real(default), dimension(2) :: m_in = 0
       logical, dimension(2) :: photon = .false.
       logical :: generate = .false.
       class(rng_factory_t), allocatable :: rng_factory
       real(default) :: sqrts = 0
       real(default) :: eps = 0
       integer :: ver = 0
       integer :: rev = 0
       character(6) :: acc = "?"
       integer :: chat = 0
     contains
       ⟨SF circe1: circe1 data: TBP⟩
    end type circe1_data_t
```

⟨*SF circe1: circe1 data: TBP*⟩≡
```
    procedure :: init => circe1_data_init
```

⟨SF circe1: procedures⟩≡
```
    subroutine circe1_data_init &
        (data, model, pdg_in, sqrts, eps, out_photon, &
         ver, rev, acc, chat)
      class(circe1_data_t), intent(out) :: data
      type(model_t), intent(in), target :: model
      type(pdg_array_t), dimension(2), intent(in) :: pdg_in
      real(default), intent(in) :: sqrts
      real(default), intent(in) :: eps
      logical, dimension(2), intent(in) :: out_photon
      character(*), intent(in) :: acc
      integer, intent(in) :: ver, rev, chat
      data%model => model
      if (any (pdg_array_get_length (pdg_in) /= 1)) then
         call msg_fatal ("CIRCE1: incoming beam particles must be unique")
      end if
      call flavor_init (data%flv_in(1), pdg_array_get (pdg_in(1), 1), model)
      call flavor_init (data%flv_in(2), pdg_array_get (pdg_in(2), 1), model)
      data%pdg_in = flavor_get_pdg (data%flv_in)
      data%m_in = flavor_get_mass (data%flv_in)
      data%sqrts = sqrts
      data%eps = eps
      data%photon = out_photon
      data%ver = ver
      data%rev = rev
      data%acc = acc
      data%chat = chat
      call data%check ()
      call circex (0.d0, 0.d0, dble (data%sqrts), &
           data%acc, data%ver, data%rev, data%chat)
    end subroutine circe1_data_init
```

Activate the generator mode. We import a RNG factory into the data type,
which can then spawn RNG generator objects.

⟨SF circe1: circe1 data: TBP⟩+≡
```
    procedure :: set_generator_mode => circe1_data_set_generator_mode
```

⟨SF circe1: procedures⟩+≡
```
    subroutine circe1_data_set_generator_mode (data, rng_factory)
      class(circe1_data_t), intent(inout) :: data
      class(rng_factory_t), intent(inout), allocatable :: rng_factory
      data%generate = .true.
      call move_alloc (from = rng_factory, to = data%rng_factory)
    end subroutine circe1_data_set_generator_mode
```

Handle error conditions.

⟨SF circe1: circe1 data: TBP⟩+≡
```
    procedure :: check => circe1_data_check
```

⟨SF circe1: procedures⟩+≡
```
    subroutine circe1_data_check (data)
      class(circe1_data_t), intent(in) :: data
      type(flavor_t) :: flv_electron, flv_photon
      call flavor_init (flv_electron, ELECTRON, data%model)
```

```
      call flavor_init (flv_photon, PHOTON, data%model)
      if (flavor_get_pdg (flv_electron) == UNDEFINED &
            .or. flavor_get_pdg (flv_photon) == UNDEFINED) then
        call msg_fatal ("CIRCE1: model must contain photon and electron")
      end if
      if (any (abs (data%pdg_in) /= ELECTRON) &
            .or. (data%pdg_in(1) /= - data%pdg_in(2))) then
        call msg_fatal ("CIRCE1: applicable only for e+e- or e-e+ collisions")
      end if
      if (data%eps <= 0) then
        call msg_error ("CIRCE1: circe1_eps = 0: integration will &
              &miss x=1 peak")
      end if
    end subroutine circe1_data_check
```

Output

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
    procedure :: write => circe1_data_write
```

⟨*SF circe1: procedures*⟩+≡
```
    subroutine circe1_data_write (data, unit, verbose)
      class(circe1_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)") "CIRCE1 data:"
      write (u, "(3x,A,2(1x,A))") "prt_in   =", &
            char (flavor_get_name (data%flv_in(1))), &
            char (flavor_get_name (data%flv_in(2)))
      write (u, "(3x,A,2(1x,L1))")  "photon   =", data%photon
      write (u, "(3x,A,L1)")        "generate = ", data%generate
      write (u, "(3x,A,2(1x," // FMT_19 // "))") "m_in     =", data%m_in
      write (u, "(3x,A," // FMT_19 // ")") "sqrts    = ", data%sqrts
      write (u, "(3x,A," // FMT_19 // ")") "eps      = ", data%eps
      write (u, "(3x,A,I0)") "ver      = ", data%ver
      write (u, "(3x,A,I0)") "rev      = ", data%rev
      write (u, "(3x,A,A)")  "acc      = ", data%acc
      write (u, "(3x,A,I0)") "chat     = ", data%chat
      if (data%generate)  call data%rng_factory%write (u)
    end subroutine circe1_data_write
```

Return true if this structure function is in generator mode. In that case, all parameters are free, otherwise bound. (We do not support mixed cases.) Default is: no generator.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
    procedure :: is_generator => circe1_data_is_generator
```

⟨*SF circe1: procedures*⟩+≡
```
    function circe1_data_is_generator (data) result (flag)
      class(circe1_data_t), intent(in) :: data
      logical :: flag
      flag = data%generate
    end function circe1_data_is_generator
```

The number of parameters is two, collinear splitting for the two beams.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: get_n_par => circe1_data_get_n_par
```

⟨*SF circe1: procedures*⟩+≡
```
function circe1_data_get_n_par (data) result (n)
  class(circe1_data_t), intent(in) :: data
  integer :: n
  n = 2
end function circe1_data_get_n_par
```

Return the outgoing particles PDG codes. This is either the incoming particle (if a photon is radiated), or the photon if that is the particle of the hard interaction. The latter is determined via the `photon` flag. There are two entries for the two beams.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: get_pdg_out => circe1_data_get_pdg_out
```

⟨*SF circe1: procedures*⟩+≡
```
subroutine circe1_data_get_pdg_out (data, pdg_out)
  class(circe1_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer :: i, n
  n = 2
  do i = 1, n
     if (data%photon(i)) then
        pdg_out(i) = PHOTON
     else
        pdg_out(i) = data%pdg_in(i)
     end if
  end do
end subroutine circe1_data_get_pdg_out
```

This variant is not inherited, it returns integers.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: get_pdg_int => circe1_data_get_pdg_int
```

⟨*SF circe1: procedures*⟩+≡
```
function circe1_data_get_pdg_int (data) result (pdg)
  class(circe1_data_t), intent(in) :: data
  integer, dimension(2) :: pdg
  integer :: i
  do i = 1, 2
     if (data%photon(i)) then
        pdg(i) = PHOTON
     else
        pdg(i) = data%pdg_in(i)
     end if
  end do
end function circe1_data_get_pdg_int
```

1253

Allocate the interaction record.

⟨SF circe1: circe1 data: TBP⟩+≡
```
procedure :: allocate_sf_int => circe1_data_allocate_sf_int
```

⟨SF circe1: procedures⟩+≡
```
subroutine circe1_data_allocate_sf_int (data, sf_int)
  class(circe1_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (circe1_t :: sf_int)
end subroutine circe1_data_allocate_sf_int
```

10.7.3 Random Number Generator for CIRCE

The CIRCE implementation now supports a generic random-number generator object that allows for a local state as a component. To support this, we must extend the abstract type provided by CIRCE and delegate the generator call to the (also abstract) RNG used by WHIZARD.

⟨SF circe1: types⟩+≡
```
type, extends (circe1_rng_t) :: rng_obj_t
  class(rng_t), allocatable :: rng
 contains
  procedure :: generate => rng_obj_generate
end type rng_obj_t
```

⟨SF circe1: procedures⟩+≡
```
subroutine rng_obj_generate (rng_obj, u)
  class(rng_obj_t), intent(inout) :: rng_obj
  real(double), intent(out) :: u
  real(default) :: x
  call rng_obj%rng%generate (x)
  u = x
end subroutine rng_obj_generate
```

10.7.4 The CIRCE1 object

This is a $2 \to 4$ interaction, where, depending on the parameters, any two of the four outgoing particles are connected to the hard interactions, the others are radiated. Knowing that all particles are colorless, we do not have to deal with color.

The flavors are sorted such that the first two particles are the incoming leptons, the next two are the radiated particles, and the last two are the partons initiating the hard interaction.

CIRCE1 does not support polarized beams explicitly. For simplicity, we nevertheless carry beam polarization through to the outgoing electrons and make the photons unpolarized.

⟨SF circe1: types⟩+≡
```
type, extends (sf_int_t) :: circe1_t
  type(circe1_data_t), pointer :: data => null ()
  real(default), dimension(2) :: x = 0
```

```
    real(default) :: f = 0
    logical, dimension(2) :: continuum = .true.
    logical, dimension(2) :: peak = .true.
    type(rng_obj_t) :: rng_obj
  contains
  ⟨SF circe1: circe1: TBP⟩
  end type circe1_t
```

Type string: has to be here, but there is no string variable on which CIRCE1 depends. Hence, a dummy routine.

⟨*SF circe1: circe1: TBP*⟩≡
```
    procedure :: type_string => circe1_type_string
```

⟨*SF circe1: procedures*⟩+≡
```
    function circe1_type_string (object) result (string)
      class(circe1_t), intent(in) :: object
      type(string_t) :: string
      if (associated (object%data)) then
         string = "CIRCE1: beamstrahlung"
      else
         string = "CIRCE1: [undefined]"
      end if
    end function circe1_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF circe1: circe1: TBP*⟩+≡
```
    procedure :: write => circe1_write
```

⟨*SF circe1: procedures*⟩+≡
```
    subroutine circe1_write (object, unit, testflag)
      class(circe1_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      if (associated (object%data)) then
         call object%data%write (u)
         if (object%data%generate)  call object%rng_obj%rng%write (u)
         if (object%status >= SF_DONE_KINEMATICS) then
            write (u, "(3x,A,2(1x," // FMT_17 // "))")  "x =", object%x
            if (object%status >= SF_FAILED_EVALUATION) then
               write (u, "(3x,A,1x," // FMT_17 // ")")  "f =", object%f
            end if
         end if
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "CIRCE1 data: [undefined]"
      end if
    end subroutine circe1_write
```

⟨*SF circe1: circe1: TBP*⟩+≡
```
    procedure :: init => circe1_init
```

⟨*SF circe1: procedures*⟩+≡

```
    subroutine circe1_init (sf_int, data)
      class(circe1_t), intent(out) :: sf_int
      class(sf_data_t), intent(in), target :: data
      logical, dimension(6) :: mask_h
      type(quantum_numbers_mask_t), dimension(6) :: mask
      integer, dimension(6) :: hel_lock
      type(polarization_t) :: pol1, pol2
      type(quantum_numbers_t), dimension(1) :: qn_fc1, qn_hel1, qn_fc2, qn_hel2
      type(flavor_t) :: flv_photon
      real(default), dimension(2) :: mi2, mr2, mo2
      type(quantum_numbers_t) :: qn_photon, qn1, qn2
      type(quantum_numbers_t), dimension(6) :: qn
      type(state_iterator_t) :: it_hel1, it_hel2
      hel_lock = 0
      mask_h = .false.
      select type (data)
      type is (circe1_data_t)
         mi2 = data%m_in**2
         if (data%photon(1)) then
            hel_lock(1) = 3;  hel_lock(3) = 1;  mask_h(5) = .true.
            mr2(1) = mi2(1)
            mo2(1) = 0._default
         else
            hel_lock(1) = 5;  hel_lock(5) = 1;  mask_h(3) = .true.
            mr2(1) = 0._default
            mo2(1) = mi2(1)
         end if
         if (data%photon(2)) then
            hel_lock(2) = 4;  hel_lock(4) = 2;  mask_h(6) = .true.
            mr2(2) = mi2(2)
            mo2(2) = 0._default
         else
            hel_lock(2) = 6;  hel_lock(6) = 2;  mask_h(4) = .true.
            mr2(2) = 0._default
            mo2(2) = mi2(2)
         end if
         mask = new_quantum_numbers_mask (.false., .false., mask_h)
         call sf_int%base_init (mask, mi2, mr2, mo2, &
              hel_lock = hel_lock)
         sf_int%data => data
         call flavor_init (flv_photon, PHOTON, data%model)
         call quantum_numbers_init (qn_photon, flv_photon)
         call polarization_init_generic (pol1, data%flv_in(1))
         call quantum_numbers_init (qn_fc1(1), flv = data%flv_in(1))
         call polarization_init_generic (pol2, data%flv_in(2))
         call quantum_numbers_init (qn_fc2(1), flv = data%flv_in(2))
         call state_iterator_init (it_hel1, pol1%state)

         do while (state_iterator_is_valid (it_hel1))
            qn_hel1 = state_iterator_get_quantum_numbers (it_hel1)
            qn1 = qn_hel1(1) .merge. qn_fc1(1)
            qn(1) = qn1
            if (data%photon(1)) then
```

```
                qn(3) = qn1;   qn(5) = qn_photon
            else
                qn(3) = qn_photon;   qn(5) = qn1
            end if
            call state_iterator_init (it_hel2, pol2%state)
            do while (state_iterator_is_valid (it_hel2))
                qn_hel2 = state_iterator_get_quantum_numbers (it_hel2)
                qn2 = qn_hel2(1) .merge. qn_fc2(1)
                qn(2) = qn2
                if (data%photon(2)) then
                    qn(4) = qn2;   qn(6) = qn_photon
                else
                    qn(4) = qn_photon;   qn(6) = qn2
                end if
                call interaction_add_state (sf_int%interaction_t, qn)
                call state_iterator_advance (it_hel2)
            end do
            call state_iterator_advance (it_hel1)
        end do
        call polarization_final (pol1)
        call polarization_final (pol2)
        call interaction_freeze (sf_int%interaction_t)
        call sf_int%set_incoming ([1,2])
        call sf_int%set_radiated ([3,4])
        call sf_int%set_outgoing ([5,6])
        sf_int%status = SF_INITIAL
    end select
    if (sf_int%data%generate) then
        call sf_int%data%rng_factory%make (sf_int%rng_obj%rng)
    end if
end subroutine circe1_init
```

10.7.5 Kinematics

Refer to the `data` component.

⟨*SF circe1: circe1: TBP*⟩+≡
```
    procedure :: is_generator => circe1_is_generator
```

⟨*SF circe1: procedures*⟩+≡
```
    function circe1_is_generator (sf_int) result (flag)
      class(circe1_t), intent(in) :: sf_int
      logical :: flag
      flag = sf_int%data%is_generator ()
    end function circe1_is_generator
```

Generate free parameters, if generator mode is on. Otherwise, the parameters
will be discarded.

⟨*SF circe1: circe1: TBP*⟩+≡
```
    procedure :: generate_free => circe1_generate_free
```

⟨*SF circe1: procedures*⟩+≡
```
    subroutine circe1_generate_free (sf_int, r, rb,  x_free)
      class(circe1_t), intent(inout) :: sf_int
```

```
        real(default), dimension(:), intent(out) :: r, rb
        real(default), intent(inout) :: x_free

        if (sf_int%data%generate) then
           call circe_generate (r, sf_int%data%get_pdg_int (), sf_int%rng_obj)
           rb = 1 - r
           x_free = x_free * product (r)
        else
           r = 0
           rb= 1
        end if
     end subroutine circe1_generate_free
```

Generator mode: depending on the particle codes, call one of the available `girce`
generators. Illegal particle code combinations should have been caught during
data initialization.

⟨*SF circe1: procedures*⟩+≡
```
     subroutine circe_generate (x, pdg, rng_obj)
       real(default), dimension(2), intent(out) :: x
       integer, dimension(2), intent(in) :: pdg
       class(rng_obj_t), intent(inout) :: rng_obj
       real(double) :: xc1, xc2
       select case (abs (pdg(1)))
       case (ELECTRON)
          select case (abs (pdg(2)))
          case (ELECTRON)
             call gircee (xc1, xc2, rng_obj = rng_obj)
          case (PHOTON)
             call girceg (xc1, xc2, rng_obj = rng_obj)
          end select
       case (PHOTON)
          select case (abs (pdg(2)))
          case (ELECTRON)
             call girceg (xc2, xc1, rng_obj = rng_obj)
          case (PHOTON)
             call gircgg (xc1, xc2, rng_obj = rng_obj)
          end select
       end select
       x = [xc1, xc2]
     end subroutine circe_generate
```

Set kinematics. The r values (either from integration or from the generator call
above) are copied to x unchanged, and f is unity. We store the x values, so we
can use them for the evaluation later.

⟨*SF circe1: circe1: TBP*⟩+≡
```
     procedure :: complete_kinematics => circe1_complete_kinematics
```

⟨*SF circe1: procedures*⟩+≡
```
     subroutine circe1_complete_kinematics (sf_int, x, f, r, rb, map)
       class(circe1_t), intent(inout) :: sf_int
       real(default), dimension(:), intent(out) :: x
       real(default), intent(out) :: f
       real(default), dimension(:), intent(in) :: r
```

1258

```
    real(default), dimension(:), intent(in) :: rb
    logical, intent(in) :: map
    real(default), dimension(2) :: xb1
    x = r
    sf_int%x = x
    f = 1
    xb1 = 1 - x
    call sf_int%split_momenta (x, xb1)
    select case (sf_int%status)
    case (SF_FAILED_KINEMATICS);  f = 0
    end select
  end subroutine circe1_complete_kinematics
```

Compute inverse kinematics. In generator mode, the r values are meaningless, but we copy them anyway.

⟨*SF circe1: circe1: TBP*⟩+≡
```
    procedure :: inverse_kinematics => circe1_inverse_kinematics
```

⟨*SF circe1: procedures*⟩+≡
```
    subroutine circe1_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
      class(circe1_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      real(default), dimension(2) :: xb1
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      r = x
      rb = 1 - x
      sf_int%x = x
      f = 1
      if (set_mom) then
         xb1 = 1 - x
         call sf_int%split_momenta (x, xb1)
         select case (sf_int%status)
         case (SF_FAILED_KINEMATICS);  f = 0
         end select
      end if
    end subroutine circe1_inverse_kinematics
```

10.7.6 CIRCE1 application

CIRCE is applied for the two beams at once. We can safely assume that no structure functions are applied before this, so the incoming particles are on-shell electrons/positrons.

The scale is ignored.

⟨*SF circe1: circe1: TBP*⟩+≡
```
    procedure :: apply => circe1_apply
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe1_apply (sf_int, scale)
  class(circe1_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default), dimension(2) :: xb
  real(double), dimension(2) :: xc
  real(double), parameter :: one = 1
  associate (data => sf_int%data)
    xc = sf_int%x
    xb = 1 - sf_int%x
    if (data%generate) then
        sf_int%f = 1
    else
        sf_int%f = 0
        if (all (sf_int%continuum)) then
           sf_int%f = circe (xc(1), xc(2), data%pdg_in(1), data%pdg_in(2))
        end if
        if (sf_int%continuum(2) .and. sf_int%peak(1)) then
           sf_int%f = sf_int%f &
                 + circe (one, xc(2), data%pdg_in(1), data%pdg_in(2)) &
                 * peak (xb(1), data%eps)
        end if
        if (sf_int%continuum(1) .and. sf_int%peak(2)) then
           sf_int%f = sf_int%f &
                 + circe (xc(1), one, data%pdg_in(1), data%pdg_in(2)) &
                 * peak (xb(2), data%eps)
        end if
        if (all (sf_int%peak)) then
           sf_int%f = sf_int%f &
                 + circe (one, one, data%pdg_in(1), data%pdg_in(2)) &
                 * peak (xb(1), data%eps) * peak (xb(2), data%eps)
        end if
    end if
  end associate
  call interaction_set_matrix_element &
      (sf_int%interaction_t, cmplx (sf_int%f, kind=default))
  sf_int%status = SF_EVALUATED
end subroutine circe1_apply
```

This is a smeared delta peak at zero, as an endpoint singularity. We choose an exponentially decreasing function, starting at zero, with integral (from 0 to 1) $1 - e^{-1/\epsilon}$. For small ϵ, this reduces to one.

⟨*SF circe1: procedures*⟩+≡

```
function peak (x, eps) result (f)
  real(default), intent(in) :: x, eps
  real(default) :: f
  f = exp (-x / eps) / eps
end function peak
```

10.7.7 Unit tests

⟨*SF circe1: public*⟩+≡

```
      public :: sf_circe1_test
```

⟨*SF circe1: tests*⟩≡
```
    subroutine sf_circe1_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF circe1: execute tests⟩
    end subroutine sf_circe1_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF circe1: execute tests*⟩≡
```
    call test (sf_circe1_1, "sf_circe1_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF circe1: tests*⟩+≡
```
    subroutine sf_circe1_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(pdg_array_t), dimension(2) :: pdg_in
      type(pdg_array_t), dimension(2) :: pdg_out
      integer, dimension(:), allocatable :: pdg1, pdg2
      class(sf_data_t), allocatable :: data

      write (u, "(A)")  "* Test output: sf_circe1_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
           &CIRCE structure function data"
      write (u, "(A)")

      write (u, "(A)")  "* Create empty data object"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QED"), &
           var_str ("QED.mdl"), os_data, model)
      pdg_in(1) = ELECTRON
      pdg_in(2) = -ELECTRON

      allocate (circe1_data_t :: data)
      call data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize"
      write (u, "(A)")

      select type (data)
      type is (circe1_data_t)
         call data%init (model, pdg_in, &
              sqrts = 500._default, &
```

1261

```
            eps = 1e-6_default, &
            out_photon = [.false., .false.], &
            ver = 0, &
            rev = 0, &
            acc = "SBAND", &
            chat = 0)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    pdg2 = pdg_out(2)
    write (u, "(2x,99(1x,I0))")  pdg1, pdg2

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_circe1_1"

  end subroutine sf_circe1_1
```

Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

⟨*SF circe1: execute tests*⟩+≡
```
    call test (sf_circe1_2, "sf_circe1_2", &
         "structure function instance", &
         u, results)
```

⟨*SF circe1: tests*⟩+≡
```
    subroutine sf_circe1_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t), dimension(2) :: flv
      type(pdg_array_t), dimension(2) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k1, k2
      type(vector4_t), dimension(4) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_circe1_2"
      write (u, "(A)")  "*   Purpose: initialize and fill &
```

1262

```
      &circe1 structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("QED"), &
      var_str ("QED.mdl"), os_data, model)
call flavor_init (flv(1), ELECTRON, model)
call flavor_init (flv(2), -ELECTRON, model)
pdg_in(1) = ELECTRON
pdg_in(2) = -ELECTRON

call reset_interaction_counter ()

allocate (circe1_data_t :: data)
select type (data)
type is (circe1_data_t)
   call data%init (model, pdg_in, &
        sqrts = 500._default, &
        eps = 1e-6_default, &
        out_photon = [.false., .false.], &
        ver = 0, &
        rev = 0, &
        acc = "SBAND", &
        chat = 0)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv(1))**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv(2))**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.95,0.85."
write (u, "(A)")

allocate (r (data%get_n_par ()))
```

```
      allocate (rb(size (r)))
      allocate (x (size (r)))

      r = [0.9_default, 0.8_default]
      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A,9(1x,F10.7))")  "x =", x
      write (u, "(A,9(1x,F10.7))")  "f =", f

      write (u, "(A)")
      write (u, "(A)")   "* Recover x from momenta"
      write (u, "(A)")

      q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
      call sf_int%final ()
      deallocate (sf_int)

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
      call sf_int%set_beam_index ([1, 2])

      call sf_int%seed_kinematics ([k1, k2])
      call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
      call sf_int%recover_x (x)

      write (u, "(A,9(1x,F10.7))")  "x =", x

      write (u, "(A)")
      write (u, "(A)")   "* Evaluate"
      write (u, "(A)")

      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%apply (scale = 0._default)
      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: sf_circe1_2"

    end subroutine sf_circe1_2
```

Generator mode

Construct and evaluate a structure function object in generator mode.

⟨*SF circe1: execute tests*⟩+≡

```
      call test (sf_circe1_3, "sf_circe1_3", &
           "generator mode", &
           u, results)
```

⟨*SF circe1: tests*⟩+≡

```
     subroutine sf_circe1_3 (u)
       integer, intent(in) :: u
       type(os_data_t) :: os_data
       type(model_list_t) :: model_list
       type(model_t), pointer :: model
       type(flavor_t), dimension(2) :: flv
       type(pdg_array_t), dimension(2) :: pdg_in
       class(sf_data_t), allocatable, target :: data
       class(rng_factory_t), allocatable :: rng_factory
       class(sf_int_t), allocatable :: sf_int
       type(vector4_t) :: k1, k2
       real(default) :: E
       real(default), dimension(:), allocatable :: r, rb, x
       real(default) :: f, x_free

       write (u, "(A)")  "* Test output: sf_circe1_3"
       write (u, "(A)")  "*   Purpose: initialize and fill &
            &circe1 structure function object"
       write (u, "(A)")

       write (u, "(A)")  "* Initialize configuration data"
       write (u, "(A)")

       call os_data_init (os_data)
       call syntax_model_file_init ()
       call model_list%read_model (var_str ("QED"), &
            var_str ("QED.mdl"), os_data, model)
       call flavor_init (flv(1), ELECTRON, model)
       call flavor_init (flv(2), -ELECTRON, model)
       pdg_in(1) = ELECTRON
       pdg_in(2) = -ELECTRON

       call reset_interaction_counter ()

       allocate (circe1_data_t :: data)
       allocate (rng_test_factory_t :: rng_factory)
       select type (data)
       type is (circe1_data_t)
          call data%init (model, pdg_in, &
               sqrts = 500._default, &
               eps = 1e-6_default, &
               out_photon = [.false., .false.], &
               ver = 0, &
               rev = 0, &
               acc = "SBAND", &
               chat = 0)
          call data%set_generator_mode (rng_factory)
       end select

       write (u, "(A)")  "* Initialize structure-function object"
```

```
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])
select type (sf_int)
type is (circe1_t)
   call sf_int%rng_obj%rng%init (3)
end select

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv(1))**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv(2))**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Generate x"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r  = 0
rb = 0
x_free = 1
call sf_int%generate_free (r, rb, x_free)
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
write (u, "(A)")

call sf_int%apply (scale = 0._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_circe1_3"
```

```
end subroutine sf_circe1_3
```

10.8 Lepton Collider Beamstrahlung and Photon collider: CIRCE2

⟨`sf_circe2.f90`⟩≡
  ⟨*File header*⟩

  module sf_circe2

  ⟨*Use kinds*⟩
    use kinds, only: double !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: FMT_19 !NODEP!
    use diagnostics !NODEP!
    use os_interface
    use unit_tests
    use lorentz !NODEP!
    use rng_base
    use selectors
    use pdg_arrays
    use models
    use flavors
    use helicities
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base
    use circe2, circe2_rng_t => rng_type !NODEP!

  ⟨*Standard module head*⟩

  ⟨*SF circe2: public*⟩

  ⟨*SF circe2: types*⟩

  contains

  ⟨*SF circe2: procedures*⟩

  ⟨*SF circe2: tests*⟩

  end module sf_circe2

10.8.1 Physics

CIRCE2 describes photon spectra Beamstrahlung is applied before ISR. The CIRCE2 implementation has a single structure function for both beams (which makes sense since it has to be switched on or off for both beams simultaneously).

10.8.2 The CIRCE2 data block

The CIRCE2 parameters are: file and collider specification, incoming (= outgoing) particles. The luminosity is returned by `cir2lm`.

⟨*SF circe2: public*⟩≡
```
public :: circe2_data_t
```

⟨*SF circe2: types*⟩≡
```
type, extends (sf_data_t) :: circe2_data_t
   private
   type(model_t), pointer :: model => null ()
   type(flavor_t), dimension(2) :: flv_in
   integer, dimension(2) :: pdg_in
   real(default) :: sqrts = 0
   logical :: polarized = .false.
   class(rng_factory_t), allocatable :: rng_factory
   type(string_t) :: filename
   type(string_t) :: file
   type(string_t) :: design
   real(default) :: lumi = 0
   real(default), dimension(4) :: lumi_hel_frac = 0
   integer, dimension(0:4) :: h1 = [0, -1, -1, 1, 1]
   integer, dimension(0:4) :: h2 = [0, -1,  1,-1, 1]
   integer :: error = 1
 contains
   ⟨SF circe2: circe2 data: TBP⟩
end type circe2_data_t
```

⟨*SF circe2: circe2 data: TBP*⟩≡
```
procedure :: init => circe2_data_init
```

⟨*SF circe2: procedures*⟩≡
```
subroutine circe2_data_init &
    (data, os_data, model, pdg_in, sqrts, polarized, file, design)
  class(circe2_data_t), intent(out) :: data
  type(os_data_t), intent(in) :: os_data
  type(model_t), intent(in), target :: model
  type(pdg_array_t), dimension(2), intent(in) :: pdg_in
  real(default), intent(in) :: sqrts
  logical, intent(in) :: polarized
  type(string_t), intent(in) :: file, design
  integer :: h
  data%model => model
  if (any (pdg_array_get_length (pdg_in) /= 1)) then
     call msg_fatal ("CIRCE2: incoming beam particles must be unique")
  end if
  call flavor_init (data%flv_in(1), pdg_array_get (pdg_in(1), 1), model)
  call flavor_init (data%flv_in(2), pdg_array_get (pdg_in(2), 1), model)
  data%pdg_in = flavor_get_pdg (data%flv_in)
  data%sqrts = sqrts
  data%polarized = polarized
  data%filename = file
  data%design = design
  call data%check_file (os_data)
  call cir2ld (trim (char(data%file)), trim (char(data%design)), &
```

```
          dble (data%sqrts), data%error)
    data%lumi = cir2lm (data%pdg_in(1), 0, data%pdg_in(2), 0)
    call data%check ()
    if (data%polarized) then
       do h = 1, 4
          data%lumi_hel_frac(h) = &
               cir2lm (data%pdg_in(1), data%h1(h), data%pdg_in(2), data%h2(h)) &
               / data%lumi
       end do
    end if
 end subroutine circe2_data_init
```

Activate the generator mode. We import a RNG factory into the data type, which can then spawn RNG generator objects.

⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: set_generator_mode => circe2_data_set_generator_mode
```

⟨*SF circe2: procedures*⟩+≡
```
    subroutine circe2_data_set_generator_mode (data, rng_factory)
      class(circe2_data_t), intent(inout) :: data
      class(rng_factory_t), intent(inout), allocatable :: rng_factory
      call move_alloc (from = rng_factory, to = data%rng_factory)
    end subroutine circe2_data_set_generator_mode
```

Check whether the requested data file is in the system directory or in the current directory.

⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: check_file => circe2_check_file
```

⟨*SF circe2: procedures*⟩+≡
```
    subroutine circe2_check_file (data, os_data)
      class(circe2_data_t), intent(inout) :: data
      type(os_data_t), intent(in) :: os_data
      logical :: exist
      type(string_t) :: file
      file = data%filename
      inquire (file = char (file), exist = exist)
      if (exist) then
         data%file = file
      else
         file = os_data%whizard_circe2path // "/" // data%filename
         inquire (file = char (file), exist = exist)
         if (exist) then
            data%file = file
         else
            call msg_fatal ("CIRCE2: data file '" // char (data%filename) &
                 // "' not found")
         end if
      end if
    end subroutine circe2_check_file
```

Handle error conditions.

⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: check => circe2_data_check
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_data_check (data)
  class(circe2_data_t), intent(in) :: data
  type(flavor_t) :: flv_photon, flv_electron
  call flavor_init (flv_photon, PHOTON, data%model)
  if (flavor_get_pdg (flv_photon) == UNDEFINED) then
     call msg_fatal ("CIRCE2: model must contain photon")
  end if
  call flavor_init (flv_electron, ELECTRON, data%model)
  if (flavor_get_pdg (flv_electron) == UNDEFINED) then
     call msg_fatal ("CIRCE2: model must contain electron")
  end if
  if (any (abs (data%pdg_in) /= PHOTON .and. abs (data%pdg_in) /= ELECTRON)) &
        then
     call msg_fatal ("CIRCE2: applicable only for e+e- or photon collisions")
  end if
  select case (data%error)
  case (-1)
     call msg_fatal ("CIRCE2: data file not found.")
  case (-2)
     call msg_fatal ("CIRCE2: beam parameters do not match data file.")
  case (-3)
     call msg_fatal ("CIRCE2: invalid format of data file.")
  case (-4)
     call msg_fatal ("CIRCE2: data file too large.")
  end select
  if (data%lumi == 0) then
     call msg_fatal ("CIRCE2: luminosity vanishes for specified beams.")
  end if
end subroutine circe2_data_check
```

Output

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
procedure :: write => circe2_data_write
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_data_write (data, unit, verbose)
  class(circe2_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, h
  u = output_unit (unit)
  write (u, "(1x,A)") "CIRCE2 data:"
  write (u, "(3x,A,A)")       "file  = ", char(data%filename)
  write (u, "(3x,A,A)")       "design = ", char(data%design)
  write (u, "(3x,A," // FMT_19 // ")") "sqrts  = ", data%sqrts
  write (u, "(3x,A,A,A,A)")   "prt_in = ", &
        char (flavor_get_name (data%flv_in(1))), &
        ", ", char (flavor_get_name (data%flv_in(2)))
  write (u, "(3x,A,L1)")      "polarized  = ", data%polarized
  write (u, "(3x,A," // FMT_19 // ")") "luminosity = ", data%lumi
  if (data%polarized) then
     do h = 1, 4
        write (u, "(6x,'(',I2,1x,I2,')',1x,'=',1x)", advance="no") &
```

```
              data%h1(h), data%h2(h)
          write (u, "(6x, " // FMT_19 // ")")  data%lumi_hel_frac(h)
        end do
     end if
     call data%rng_factory%write (u)
   end subroutine circe2_data_write
```

This is always in generator mode.
⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: is_generator => circe2_data_is_generator
```
⟨*SF circe2: procedures*⟩+≡
```
    function circe2_data_is_generator (data) result (flag)
      class(circe2_data_t), intent(in) :: data
      logical :: flag
      flag = .true.
    end function circe2_data_is_generator
```

The number of parameters is two, collinear splitting for the two beams.
⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: get_n_par => circe2_data_get_n_par
```
⟨*SF circe2: procedures*⟩+≡
```
    function circe2_data_get_n_par (data) result (n)
      class(circe2_data_t), intent(in) :: data
      integer :: n
      n = 2
    end function circe2_data_get_n_par
```

Return the outgoing particles PDG codes. They are equal to the incoming ones.
⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: get_pdg_out => circe2_data_get_pdg_out
```
⟨*SF circe2: procedures*⟩+≡
```
    subroutine circe2_data_get_pdg_out (data, pdg_out)
      class(circe2_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      integer :: i, n
      n = 2
      do i = 1, n
         pdg_out(i) = data%pdg_in(i)
      end do
    end subroutine circe2_data_get_pdg_out
```

Allocate the interaction record.
⟨*SF circe2: circe2 data: TBP*⟩+≡
```
    procedure :: allocate_sf_int => circe2_data_allocate_sf_int
```
⟨*SF circe2: procedures*⟩+≡
```
    subroutine circe2_data_allocate_sf_int (data, sf_int)
      class(circe2_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
      allocate (circe2_t :: sf_int)
    end subroutine circe2_data_allocate_sf_int
```

10.8.3 Random Number Generator for CIRCE

The CIRCE implementation now supports a generic random-number generator object that allows for a local state as a component. To support this, we must extend the abstract type provided by CIRCE and delegate the generator call to the (also abstract) RNG used by WHIZARD.

⟨*SF circe2: types*⟩+≡
```
type, extends (circe2_rng_t) :: rng_obj_t
  class(rng_t), allocatable :: rng
 contains
  procedure :: generate => rng_obj_generate
end type rng_obj_t
```

⟨*SF circe2: procedures*⟩+≡
```
subroutine rng_obj_generate (rng_obj, u)
  class(rng_obj_t), intent(inout) :: rng_obj
  real(double), intent(out) :: u
  real(default) :: x
  call rng_obj%rng%generate (x)
  u = x
end subroutine rng_obj_generate
```

10.8.4 The CIRCE2 object

For CIRCE2 spectra it does not make sense to describe the state matrix as a radiation interaction, even if photons originate from laser backscattering. Instead, it is a $2 \to 2$ interaction where the incoming particles are identical to the outgoing ones.

The current implementation of CIRCE2 does support polarization and classical correlations, but no entanglement, so the density matrix of the outgoing particles is diagonal. The incoming particles are unpolarized (user-defined polarization for beams is meaningless, since polarization is described by the data file). The outgoing particles are polarized or polarization-averaged, depending on user request.

When assigning matrix elements, we scan the previously initialized state matrix. For each entry, we extract helicity and call the structure function. In the unpolarized case, the helicity is undefined and replaced by value zero. In the polarized case, there are four entries. If the generator is used, only one entry is nonzero in each call. Which one, is determined by comparing with a previously (randomly, distributed by relative luminosity) selected pair of helicities.

⟨*SF circe2: types*⟩+≡
```
type, extends (sf_int_t) :: circe2_t
  type(circe2_data_t), pointer :: data => null ()
  type(rng_obj_t) :: rng_obj
  type(selector_t) :: selector
  integer :: h_sel = 0
 contains
 ⟨SF circe2: circe2: TBP⟩
end type circe2_t
```

Type string: show file and design of `CIRCE2` structure function.

⟨*SF circe2: circe2: TBP*⟩≡
```
procedure :: type_string => circe2_type_string
```

⟨*SF circe2: procedures*⟩+≡
```
function circe2_type_string (object) result (string)
  class(circe2_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "CIRCE2: " // object%data%design
  else
     string = "CIRCE2: [undefined]"
  end if
end function circe2_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF circe2: circe2: TBP*⟩+≡
```
procedure :: write => circe2_write
```

⟨*SF circe2: procedures*⟩+≡
```
subroutine circe2_write (object, unit, testflag)
  class(circe2_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "CIRCE2 data: [undefined]"
  end if
end subroutine circe2_write
```

⟨*SF circe2: circe2: TBP*⟩+≡
```
procedure :: init => circe2_init
```

⟨*SF circe2: procedures*⟩+≡
```
subroutine circe2_init (sf_int, data)
  class(circe2_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  logical, dimension(4) :: mask_h
  real(default), dimension(0) :: null_array
  type(quantum_numbers_mask_t), dimension(4) :: mask
  type(quantum_numbers_t), dimension(4) :: qn
  type(helicity_t) :: hel
  integer :: h
  select type (data)
  type is (circe2_data_t)
     mask_h(1:2) = .true.
     mask_h(3:4) = .not. data%polarized
     mask = new_quantum_numbers_mask (.false., .false., mask_h)
     call sf_int%base_init (mask, [0._default, 0._default], &
           null_array, [0._default, 0._default])
```

1274

```
        sf_int%data => data
        if (data%polarized) then
           call sf_int%selector%init (data%lumi_hel_frac)
        end if
        call quantum_numbers_init (qn(1), flv = data%flv_in(1))
        call quantum_numbers_init (qn(2), flv = data%flv_in(2))
        if (data%polarized) then
           do h = 1, 4
              call helicity_init (hel, data%h1(h))
              call quantum_numbers_init (qn(3), flv = data%flv_in(1), hel = hel)
              call helicity_init (hel, data%h2(h))
              call quantum_numbers_init (qn(4), flv = data%flv_in(2), hel = hel)
              call interaction_add_state (sf_int%interaction_t, qn)
           end do
        else
           call quantum_numbers_init (qn(3), flv = data%flv_in(1))
           call quantum_numbers_init (qn(4), flv = data%flv_in(2))
           call interaction_add_state (sf_int%interaction_t, qn)
        end if
        call interaction_freeze (sf_int%interaction_t)
        call sf_int%set_incoming ([1,2])
        call sf_int%set_outgoing ([3,4])
        call sf_int%data%rng_factory%make (sf_int%rng_obj%rng)
        sf_int%status = SF_INITIAL
     end select
   end subroutine circe2_init
```

10.8.5   Kinematics

Refer to the `data` component.

⟨*SF circe2: circe2: TBP*⟩+≡
```
    procedure :: is_generator => circe2_is_generator
```

⟨*SF circe2: procedures*⟩+≡
```
    function circe2_is_generator (sf_int) result (flag)
      class(circe2_t), intent(in) :: sf_int
      logical :: flag
      flag = sf_int%data%is_generator ()
    end function circe2_is_generator
```

Generate free parameters. We first select a helicity, which we have to store, then generate x values for that helicity.

⟨*SF circe2: circe2: TBP*⟩+≡
```
    procedure :: generate_free => circe2_generate_free
```

⟨*SF circe2: procedures*⟩+≡
```
    subroutine circe2_generate_free (sf_int, r, rb, x_free)
      class(circe2_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: r, rb
      real(default), intent(inout) :: x_free
      integer :: h_sel
      if (sf_int%data%polarized) then
         call sf_int%selector%generate (sf_int%rng_obj%rng, h_sel)
```

```
        else
           h_sel = 0
        end if
        sf_int%h_sel = h_sel
        call circe2_generate (r, sf_int%data%pdg_in, &
             [sf_int%data%h1(h_sel), sf_int%data%h2(h_sel)], &
             sf_int%rng_obj)
        rb = 1 - r
        x_free = x_free * product (r)
     end subroutine circe2_generate_free
```

Generator mode: call the CIRCE2 generator for the given particles and helicities. (For unpolarized generation, helicities are zero.)

⟨*SF circe2: procedures*⟩+≡
```
     subroutine circe2_generate (x, pdg, hel, rng_obj)
       real(default), dimension(2), intent(out) :: x
       integer, dimension(2), intent(in) :: pdg
       integer, dimension(2), intent(in) :: hel
       class(rng_obj_t), intent(inout) :: rng_obj
       real(double) :: xc1, xc2
       call cir2gn (pdg(1), hel(1), pdg(2), hel(2), xc1, xc2, rng_obj = rng_obj)
       x = [xc1, xc2]
     end subroutine circe2_generate
```

Set kinematics. Trivial here.

⟨*SF circe2: circe2: TBP*⟩+≡
```
     procedure :: complete_kinematics => circe2_complete_kinematics
```

⟨*SF circe2: procedures*⟩+≡
```
     subroutine circe2_complete_kinematics (sf_int, x, f, r, rb, map)
       class(circe2_t), intent(inout) :: sf_int
       real(default), dimension(:), intent(out) :: x
       real(default), intent(out) :: f
       real(default), dimension(:), intent(in) :: r
       real(default), dimension(:), intent(in) :: rb
       logical, intent(in) :: map
       if (map) then
          call msg_fatal ("CIRCE2: map flag not supported")
       else
          x = r
          f = 1
       end if
       call sf_int%reduce_momenta (x)
     end subroutine circe2_complete_kinematics
```

Compute inverse kinematics.

⟨*SF circe2: circe2: TBP*⟩+≡
```
     procedure :: inverse_kinematics => circe2_inverse_kinematics
```

⟨*SF circe2: procedures*⟩+≡
```
     subroutine circe2_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
       class(circe2_t), intent(inout) :: sf_int
       real(default), dimension(:), intent(in) :: x
```

```
real(default), intent(out) :: f
real(default), dimension(:), intent(out) :: r
real(default), dimension(:), intent(out) :: rb
logical, intent(in) :: map
logical, intent(in), optional :: set_momenta
logical :: set_mom
set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
if (map) then
   call msg_fatal ("CIRCE2: map flag not supported")
else
   r = x
   rb= 1 - r
   f = 1
end if
if (set_mom) then
   call sf_int%reduce_momenta (x)
end if
end subroutine circe2_inverse_kinematics
```

10.8.6   CIRCE2 application

This function works on both beams. In polarized mode, we set only the selected helicity. In unpolarized mode, the interaction has only one entry, and the factor is unity.

⟨*SF circe2: circe2: TBP*⟩+≡
```
procedure :: apply => circe2_apply
```

⟨*SF circe2: procedures*⟩+≡
```
subroutine circe2_apply (sf_int, scale)
  class(circe2_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  complex(default) :: f
  integer :: h
  associate (data => sf_int%data)
    f = 1
    if (data%polarized) then
       h = sf_int%h_sel
    else
       h = 1
    end if
    call interaction_set_matrix_element (sf_int%interaction_t, h, f)
  end associate
  sf_int%status = SF_EVALUATED
end subroutine circe2_apply
```

10.8.7   Unit tests

⟨*SF circe2: public*⟩+≡
```
public :: sf_circe2_test
```

⟨*SF circe2: tests*⟩≡
```
    subroutine sf_circe2_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF circe2: execute tests⟩
    end subroutine sf_circe2_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF circe2: execute tests*⟩≡
```
    call test (sf_circe2_1, "sf_circe2_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF circe2: tests*⟩+≡
```
    subroutine sf_circe2_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(pdg_array_t), dimension(2) :: pdg_in
      type(pdg_array_t), dimension(2) :: pdg_out
      integer, dimension(:), allocatable :: pdg1, pdg2
      class(sf_data_t), allocatable :: data
      class(rng_factory_t), allocatable :: rng_factory

      write (u, "(A)")  "* Test output: sf_circe2_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
           &CIRCE structure function data"
      write (u, "(A)")

      write (u, "(A)")  "* Create empty data object"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QED"), &
           var_str ("QED.mdl"), os_data, model)
      pdg_in(1) = PHOTON
      pdg_in(2) = PHOTON

      allocate (circe2_data_t :: data)
      allocate (rng_test_factory_t :: rng_factory)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize (unpolarized)"
      write (u, "(A)")

      select type (data)
      type is (circe2_data_t)
         call data%init (os_data, model, pdg_in, &
              sqrts = 500._default, &
```

```
            polarized = .false., &
            file = var_str ("teslagg_500_polavg.circe"), &
            design = var_str ("TESLA/GG"))
      call data%set_generator_mode (rng_factory)
   end select

   call data%write (u)

   write (u, "(A)")

   write (u, "(1x,A)")  "Outgoing particle codes:"
   call data%get_pdg_out (pdg_out)
   pdg1 = pdg_out(1)
   pdg2 = pdg_out(2)
   write (u, "(2x,99(1x,I0))")  pdg1, pdg2

   write (u, "(A)")
   write (u, "(A)")  "* Initialize (polarized)"
   write (u, "(A)")

   allocate (rng_test_factory_t :: rng_factory)

   select type (data)
   type is (circe2_data_t)
      call data%init (os_data, model, pdg_in, &
            sqrts = 500._default, &
            polarized = .true., &
            file = var_str ("teslagg_500.circe"), &
            design = var_str ("TESLA/GG"))
      call data%set_generator_mode (rng_factory)
   end select

   call data%write (u)

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: sf_circe2_1"

end subroutine sf_circe2_1
```

Generator mode, unpolarized

Construct and evaluate a structure function object in generator mode.

⟨*SF circe2: execute tests*⟩+≡
```
   call test (sf_circe2_2, "sf_circe2_2", &
        "generator, unpolarized", &
        u, results)
```

⟨*SF circe2: tests*⟩+≡
```
   subroutine sf_circe2_2 (u)
     integer, intent(in) :: u
```

```
type(os_data_t) :: os_data
type(model_list_t) :: model_list
type(model_t), pointer :: model
type(flavor_t), dimension(2) :: flv
type(pdg_array_t), dimension(2) :: pdg_in
class(sf_data_t), allocatable, target :: data
class(rng_factory_t), allocatable :: rng_factory
class(sf_int_t), allocatable :: sf_int
type(vector4_t) :: k1, k2
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: f, x_free

write (u, "(A)")  "* Test output: sf_circe2_2"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &circe2 structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("QED"), &
     var_str ("QED.mdl"), os_data, model)
call flavor_init (flv(1), PHOTON, model)
call flavor_init (flv(2), PHOTON, model)
pdg_in(1) = PHOTON
pdg_in(2) = PHOTON

call reset_interaction_counter ()

allocate (circe2_data_t :: data)
allocate (rng_test_factory_t :: rng_factory)
select type (data)
type is (circe2_data_t)
   call data%init (os_data, model, pdg_in, &
        sqrts = 500._default, &
        polarized = .false., &
        file = var_str ("teslagg_500_polavg.circe"), &
        design = var_str ("TESLA/GG"))
   call data%set_generator_mode (rng_factory)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])
select type (sf_int)
type is (circe2_t)
   call sf_int%rng_obj%rng%init (3)
end select
```

```
            write (u, "(A)")  "* Initialize incoming momentum with E=500"
            write (u, "(A)")
            E = 250
            k1 = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv(1))**2), 3)
            k2 = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv(2))**2), 3)
            call vector4_write (k1, u)
            call vector4_write (k2, u)
            call sf_int%seed_kinematics ([k1, k2])

            write (u, "(A)")
            write (u, "(A)")  "* Generate x"
            write (u, "(A)")

            allocate (r (data%get_n_par ()))
            allocate (rb(size (r)))
            allocate (x (size (r)))

            r  = 0
            rb = 0
            x_free = 1
            call sf_int%generate_free (r, rb, x_free)
            call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

            write (u, "(A,9(1x,F10.7))")  "x =", x
            write (u, "(A,9(1x,F10.7))")  "f =", f
            write (u, "(A,9(1x,F10.7))")  "xf=", x_free

            write (u, "(A)")
            write (u, "(A)")  "* Evaluate"
            write (u, "(A)")

            call sf_int%apply (scale = 0._default)
            call sf_int%write (u)

            write (u, "(A)")
            write (u, "(A)")  "* Cleanup"

            call sf_int%final ()
            call model_list%final ()
            call syntax_model_file_final ()

            write (u, "(A)")
            write (u, "(A)")  "* Test output end: sf_circe2_2"

          end subroutine sf_circe2_2
```

Generator mode, polarized

Construct and evaluate a structure function object in generator mode.

⟨*SF circe2: execute tests*⟩+≡
```
    call test (sf_circe2_3, "sf_circe2_3", &
```

```
            "generator, polarized", &
            u, results)
```
⟨*SF circe2: tests*⟩+≡
```
    subroutine sf_circe2_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t), dimension(2) :: flv
      type(pdg_array_t), dimension(2) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(rng_factory_t), allocatable :: rng_factory
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k1, k2
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f, x_free

      write (u, "(A)")  "* Test output: sf_circe2_3"
      write (u, "(A)")  "*   Purpose: initialize and fill &
          &circe2 structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QED"), &
          var_str ("QED.mdl"), os_data, model)
      call flavor_init (flv(1), PHOTON, model)
      call flavor_init (flv(2), PHOTON, model)
      pdg_in(1) = PHOTON
      pdg_in(2) = PHOTON

      call reset_interaction_counter ()

      allocate (circe2_data_t :: data)
      allocate (rng_test_factory_t :: rng_factory)
      select type (data)
      type is (circe2_data_t)
         call data%init (os_data, model, pdg_in, &
             sqrts = 500._default, &
             polarized = .true., &
             file = var_str ("teslagg_500.circe"), &
             design = var_str ("TESLA/GG"))
         call data%set_generator_mode (rng_factory)
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
```

```
call sf_int%set_beam_index ([1,2])
select type (sf_int)
type is (circe2_t)
   call sf_int%rng_obj%rng%init (3)
end select

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv(1))**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv(2))**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Generate x"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r  = 0
rb = 0
x_free = 1
call sf_int%generate_free (r, rb, x_free)
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
write (u, "(A)")

call sf_int%apply (scale = 0._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: sf_circe2_3"

end subroutine sf_circe2_3
```

10.9 Energy-scan spectrum

This spectrum is actually a trick that allows us to plot the c.m. energy dependence of a cross section without scanning the input energy. We start with the observation that a spectrum $f(x)$, applied to one of the incoming beams only, results in a cross section

$$\sigma = \int dx\, f(x)\, \hat{\sigma}(xs). \tag{10.71}$$

We want to compute the distribution of $E = \sqrt{\hat{s}} = \sqrt{xs}$, i.e.,

$$\frac{d\sigma}{dE} = \frac{2\sqrt{x}}{\sqrt{s}} \frac{d\sigma}{dx} = \frac{2\sqrt{x}}{\sqrt{s}}\, f(x)\, \hat{\sigma}(xs), \tag{10.72}$$

so if we set

$$f(x) = \frac{\sqrt{s}}{2\sqrt{x}}, \tag{10.73}$$

we get the distribution

$$\frac{d\sigma}{dE} = \hat{\sigma}(\hat{s} = E^2). \tag{10.74}$$

We implement this as a spectrum with a single parameter x. The parameters for the individual beams are computed as $x_i = \sqrt{x}$, so they are equal and the kinematics is always symmetric.

⟨**sf_escan.f90**⟩≡
  ⟨*File header*⟩

```
  module sf_escan
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use limits, only: FMT_12 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use os_interface
    use unit_tests
    use pdg_arrays
    use models
    use flavors
    use helicities
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base
```

  ⟨*Standard module head*⟩

  ⟨*SF escan: public*⟩

  ⟨*SF escan: types*⟩

```
contains

⟨SF escan: procedures⟩

⟨SF escan: tests⟩

end module sf_escan
```

10.9.1   Data type

The `norm` is unity if the total cross section should be normalized to one, and $\sqrt{s}$ if it should be normalized to the total energy. In the latter case, the differential distribution $d\sigma/d\sqrt{\hat{s}}$ coincides with the partonic cross section $\hat{\sigma}$ as a function of $\sqrt{\hat{s}}$.

⟨*SF escan: public*⟩≡
```
    public :: escan_data_t
```

⟨*SF escan: types*⟩≡
```
    type, extends(sf_data_t) :: escan_data_t
      private
      type(flavor_t), dimension(:,:), allocatable :: flv_in
      integer, dimension(2) :: n_flv = 0
      real(default) :: norm = 1
     contains
     ⟨SF escan: escan data: TBP⟩
    end type escan_data_t
```

⟨*SF escan: escan data: TBP*⟩≡
```
    procedure :: init => escan_data_init
```

⟨*SF escan: procedures*⟩≡
```
    subroutine escan_data_init (data, model, pdg_in, norm)
      class(escan_data_t), intent(out) :: data
      type(model_t), intent(in), target :: model
      type(pdg_array_t), dimension(2), intent(in) :: pdg_in
      real(default), intent(in), optional :: norm
      real(default), dimension(2) :: m2
      integer :: i, j
      data%n_flv = pdg_array_get_length (pdg_in)
      allocate (data%flv_in (maxval (data%n_flv), 2))
      do i = 1, 2
         do j = 1, data%n_flv(i)
            call flavor_init (data%flv_in(j, i), pdg_array_get (pdg_in(i), j), &
                 model)
         end do
      end do
      m2 = flavor_get_mass (data%flv_in(1,:))
      do i = 1, 2
         if (any (flavor_get_mass (data%flv_in(1:data%n_flv(i),i)) /= m2(i))) then
            call msg_fatal ("Energy scan: incoming particle mass must be uniform")
         end if
      end do
      if (present (norm))  data%norm = norm
```

```
      end subroutine escan_data_init
```

Output

⟨*SF escan: escan data: TBP*⟩+≡
```
    procedure :: write => escan_data_write
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_data_write (data, unit, verbose)
      class(escan_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u, i, j
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)") "Energy-scan data:"
      write (u, "(3x,A)", advance="no")  "prt_in = "
      do i = 1, 2
         if (i > 1)  write (u, "(',',1x)", advance="no")
         do j = 1, data%n_flv(i)
            if (j > 1)  write (u, "(':')", advance="no")
            write (u, "(A)", advance="no") &
                 char (flavor_get_name (data%flv_in(j,i)))
         end do
      end do
      write (u, *)
      write (u, "(3x,A," // FMT_12 // ")")  "norm   =", data%norm
    end subroutine escan_data_write
```

Kinematics is completely collinear, hence there is only one parameter for a pair spectrum.

⟨*SF escan: escan data: TBP*⟩+≡
```
    procedure :: get_n_par => escan_data_get_n_par
```

⟨*SF escan: procedures*⟩+≡
```
    function escan_data_get_n_par (data) result (n)
      class(escan_data_t), intent(in) :: data
      integer :: n
      n = 1
    end function escan_data_get_n_par
```

Return the outgoing particles PDG codes. This is always the same as the incoming particle, where we use two indices for the two beams.

⟨*SF escan: escan data: TBP*⟩+≡
```
    procedure :: get_pdg_out => escan_data_get_pdg_out
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_data_get_pdg_out (data, pdg_out)
      class(escan_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      integer :: i, n
      n = 2
      do i = 1, n
         pdg_out(i) = flavor_get_pdg (data%flv_in(1:data%n_flv(i),i))
      end do
```

```
      end subroutine escan_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF escan: escan data: TBP*⟩+≡
```
    procedure :: allocate_sf_int => escan_data_allocate_sf_int
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_data_allocate_sf_int (data, sf_int)
      class(escan_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
      allocate (escan_t :: sf_int)
    end subroutine escan_data_allocate_sf_int
```

10.9.2 The Energy-scan object

This is a spectrum, not a radiation. We create an interaction with two incoming and two outgoing particles, flavor, color, and helicity being carried through. x nevertheless is only one-dimensional, as we are always using only one beam parameter.

⟨*SF escan: types*⟩+≡
```
    type, extends (sf_int_t) :: escan_t
      type(escan_data_t), pointer :: data => null ()
    contains
    ⟨SF escan: escan: TBP⟩
    end type escan_t
```

Type string: for the energy scan this is just a dummy function.

⟨*SF escan: escan: TBP*⟩≡
```
    procedure :: type_string => escan_type_string
```

⟨*SF escan: procedures*⟩+≡
```
    function escan_type_string (object) result (string)
      class(escan_t), intent(in) :: object
      type(string_t) :: string
      if (associated (object%data)) then
         string = "Escan: energy scan"
      else
         string = "Escan: [undefined]"
      end if
    end function escan_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF escan: escan: TBP*⟩+≡
```
    procedure :: write => escan_write
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_write (object, unit, testflag)
      class(escan_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
```

```
      u = output_unit (unit)
      if (associated (object%data)) then
         call object%data%write (u)
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "Energy scan data: [undefined]"
      end if
    end subroutine escan_write
```

⟨*SF escan: escan: TBP*⟩+≡
```
    procedure :: init => escan_init
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_init (sf_int, data)
      class(escan_t), intent(out) :: sf_int
      class(sf_data_t), intent(in), target :: data
      type(quantum_numbers_mask_t), dimension(4) :: mask
      integer, dimension(4) :: hel_lock
      real(default), dimension(2) :: m2
      real(default), dimension(0) :: mr2
      type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
      type(polarization_t) :: pol1, pol2
      type(state_iterator_t) :: it_hel1, it_hel2
      integer :: j1, j2
      select type (data)
      type is (escan_data_t)
         hel_lock = [3, 4, 1, 2]
         m2 = flavor_get_mass (data%flv_in(1,:))
         call sf_int%base_init (mask, m2, mr2, m2, hel_lock = hel_lock)
         sf_int%data => data
         do j1 = 1, data%n_flv(1)
            call quantum_numbers_init (qn_fc(1), &
                 flv = data%flv_in(j1,1), &
                 col = color_from_flavor (data%flv_in(j1,1)))
            call quantum_numbers_init (qn_fc(3), &
                 flv = data%flv_in(j1,1), &
                 col = color_from_flavor (data%flv_in(j1,1)))
            call polarization_init_generic (pol1, data%flv_in(j1,1))
            do j2 = 1, data%n_flv(2)
               call quantum_numbers_init (qn_fc(2), &
                    flv = data%flv_in(j2,2), &
                    col = color_from_flavor (data%flv_in(j2,2)))
               call quantum_numbers_init (qn_fc(4), &
                    flv = data%flv_in(j2,2), &
                    col = color_from_flavor (data%flv_in(j2,2)))
               call polarization_init_generic (pol2, data%flv_in(j2,2))
               call state_iterator_init (it_hel1, pol1%state)
               do while (state_iterator_is_valid (it_hel1))
                  qn_hel(1:1) = state_iterator_get_quantum_numbers (it_hel1)
                  qn_hel(3:3) = state_iterator_get_quantum_numbers (it_hel1)
                  call state_iterator_init (it_hel2, pol2%state)
                  do while (state_iterator_is_valid (it_hel2))
                     qn_hel(2:2) = state_iterator_get_quantum_numbers (it_hel2)
                     qn_hel(4:4) = state_iterator_get_quantum_numbers (it_hel2)
```

```
                qn = qn_hel .merge. qn_fc
                call interaction_add_state (sf_int%interaction_t, qn)
                call state_iterator_advance (it_hel2)
             end do
             call state_iterator_advance (it_hel1)
          end do
          call polarization_final (pol2)
       end do
       call polarization_final (pol1)
    end do
    call sf_int%set_incoming ([1,2])
    call sf_int%set_outgoing ([3,4])
    call interaction_freeze (sf_int%interaction_t)
    sf_int%status = SF_INITIAL
  end select
end subroutine escan_init
```

10.9.3   Kinematics

Set kinematics. We have a single parameter, but reduce both beams. The `map`
flag is ignored.

⟨*SF escan: escan: TBP*⟩+≡
```
    procedure :: complete_kinematics => escan_complete_kinematics
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_complete_kinematics (sf_int, x, f, r, rb, map)
      class(escan_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(out) :: f
      real(default) :: sqrt_x
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      x = r
      sqrt_x = sqrt (x(1))
      if (sqrt_x > 0) then
         f = 1 / (2 * sqrt_x)
      else
         f = 0
         sf_int%status = SF_FAILED_KINEMATICS
         return
      end if
      call sf_int%reduce_momenta ([sqrt_x, sqrt_x])
    end subroutine escan_complete_kinematics
```

Recover x. The base procedure should return two momentum fractions for
the two beams, while we have only one parameter. This is the product of the
extracted momentum fractions.

⟨*SF escan: escan: TBP*⟩+≡
```
    procedure :: recover_x => escan_recover_x
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_recover_x (sf_int, x, x_free)
      class(escan_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(inout), optional :: x_free
      real(default), dimension(2) :: xi
      call sf_int%base_recover_x (xi, x_free)
      x = product (xi)
    end subroutine escan_recover_x
```

Compute inverse kinematics.

⟨*SF escan: escan: TBP*⟩+≡
```
    procedure :: inverse_kinematics => escan_inverse_kinematics
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
      class(escan_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      real(default) :: sqrt_x
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      sqrt_x = sqrt (x(1))
      if (sqrt_x > 0) then
         f = 1 / (2 * sqrt_x)
      else
         f = 0
         sf_int%status = SF_FAILED_KINEMATICS
         return
      end if
      r = x
      rb = 1 - r
      if (set_mom) then
         call sf_int%reduce_momenta ([sqrt_x, sqrt_x])
      end if
    end subroutine escan_inverse_kinematics
```

10.9.4 Energy scan application

Here, we insert the predefined norm.

⟨*SF escan: escan: TBP*⟩+≡
```
    procedure :: apply => escan_apply
```

⟨*SF escan: procedures*⟩+≡
```
    subroutine escan_apply (sf_int, scale)
      class(escan_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
      real(default) :: f
      associate (data => sf_int%data)
```

```
      f = data%norm
    end associate
    call interaction_set_matrix_element (sf_int%interaction_t, &
         cmplx (f, kind=default))
    sf_int%status = SF_EVALUATED
  end subroutine escan_apply
```

10.9.5   Unit tests

⟨*SF escan: public*⟩+≡
```
    public :: sf_escan_test
```

⟨*SF escan: tests*⟩≡
```
    subroutine sf_escan_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*SF escan: execute tests*⟩
```
    end subroutine sf_escan_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF escan: execute tests*⟩≡
```
    call test (sf_escan_1, "sf_escan_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF escan: tests*⟩+≡
```
    subroutine sf_escan_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(pdg_array_t), dimension(2) :: pdg_in
      type(pdg_array_t), dimension(2) :: pdg_out
      integer, dimension(:), allocatable :: pdg1, pdg2
      class(sf_data_t), allocatable :: data

      write (u, "(A)")  "* Test output: sf_escan_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
          &energy-scan structure function data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QED"), &
          var_str ("QED.mdl"), os_data, model)
      pdg_in(1) = ELECTRON
      pdg_in(2) = -ELECTRON

      allocate (escan_data_t :: data)
      select type (data)
```

```
      type is (escan_data_t)
         call data%init (model, pdg_in, norm = 2._default)
      end select

      call data%write (u)

      write (u, "(A)")

      write (u, "(1x,A)")  "Outgoing particle codes:"
      call data%get_pdg_out (pdg_out)
      pdg1 = pdg_out(1)
      pdg2 = pdg_out(2)
      write (u, "(2x,99(1x,I0))")  pdg1, pdg2

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_escan_1"

    end subroutine sf_escan_1
```

g@

Probe the structure-function object

Active the beam event reader, generate an event.

⟨*SF escan: execute tests*⟩+≡
```
    call test (sf_escan_2, "sf_escan_2", &
         "generate event", &
         u, results)
```

⟨*SF escan: tests*⟩+≡
```
    subroutine sf_escan_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t), dimension(2) :: flv
      type(pdg_array_t), dimension(2) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k1, k2
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: x_free, f

      write (u, "(A)")  "* Test output: sf_escan_2"
      write (u, "(A)")  "*   Purpose: initialize and display &
           &beam-events structure function data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
```

```
call model_list%read_model (var_str ("QED"), &
     var_str ("QED.mdl"), os_data, model)
call flavor_init (flv(1), ELECTRON, model)
call flavor_init (flv(2), -ELECTRON, model)
pdg_in(1) = ELECTRON
pdg_in(2) = -ELECTRON

call reset_interaction_counter ()

allocate (escan_data_t :: data)
select type (data)
type is (escan_data_t)
   call data%init (model, pdg_in)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv(1))**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv(2))**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Set dummy parameters and generate x"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r  = 0.8
rb = 1 - r
x_free = 1

call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")
```

```
        call sf_int%recover_x (x, x_free)
        call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)

        write (u, "(A,9(1x,F10.7))")  "r =", r
        write (u, "(A,9(1x,F10.7))")  "rb=", rb
        write (u, "(A,9(1x,F10.7))")  "x =", x
        write (u, "(A,9(1x,F10.7))")  "f =", f
        write (u, "(A,9(1x,F10.7))")  "xf=", x_free

        write (u, "(A)")
        write (u, "(A)")  "* Evaluate"
        write (u, "(A)")

        call sf_int%apply (scale = 0._default)
        call sf_int%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call sf_int%final ()
        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: sf_escan_2"

    end subroutine sf_escan_2
```

10.10 Using beam event data

Instead of an analytic beam description, beam data may be provided in form of
an event file. In its most simple form, the event file contains pairs of x values,
relative to nominal beam energies. More advanced formats may include polar-
ization, etc. The current implementation carries beam polarization through, if
specified.

The code is very similar to the energy scan described above.

⟨sf_beam_events.f90⟩≡
  ⟨*File header*⟩

  module sf_beam_events

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use os_interface
    use unit_tests
    use pdg_arrays
    use models
```

```
        use flavors
        use helicities
        use colors
        use quantum_numbers
        use state_matrices
        use polarizations
        use interactions
        use sf_aux
        use sf_base
```

⟨*Standard module head*⟩

⟨*SF beam events: public*⟩

⟨*SF beam events: types*⟩

```
    contains
```

⟨*SF beam events: procedures*⟩

⟨*SF beam events: tests*⟩

```
    end module sf_beam_events
```

10.10.1 Data type

⟨*SF beam events: public*⟩≡
```
        public :: beam_events_data_t
```

⟨*SF beam events: types*⟩≡
```
        type, extends(sf_data_t) :: beam_events_data_t
          private
          type(flavor_t), dimension(2) :: flv_in
          type(string_t) :: dir
          type(string_t) :: file
          integer :: unit = 0
          logical :: warn_eof = .true.
         contains
        ⟨SF beam events: beam events data: TBP⟩
        end type beam_events_data_t
```

⟨*SF beam events: beam events data: TBP*⟩≡
```
        procedure :: init => beam_events_data_init
```

⟨*SF beam events: procedures*⟩≡
```
        subroutine beam_events_data_init (data, model, pdg_in, dir, file, warn_eof)
          class(beam_events_data_t), intent(out) :: data
          type(model_t), intent(in), target :: model
          type(pdg_array_t), dimension(2), intent(in) :: pdg_in
          type(string_t), intent(in) :: dir
          type(string_t), intent(in) :: file
          logical, intent(in), optional :: warn_eof
          if (any (pdg_array_get_length (pdg_in) /= 1)) then
             call msg_fatal ("Beam events: incoming beam particles must be unique")
```

```
          end if
          call flavor_init (data%flv_in(1), pdg_array_get (pdg_in(1), 1), model)
          call flavor_init (data%flv_in(2), pdg_array_get (pdg_in(2), 1), model)
          data%dir = dir
          data%file = file
          if (present (warn_eof))  data%warn_eof = warn_eof
        end subroutine beam_events_data_init
```

Return true since this spectrum is always in generator mode.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
        procedure :: is_generator => beam_events_data_is_generator
```

⟨*SF beam events: procedures*⟩+≡
```
        function beam_events_data_is_generator (data) result (flag)
          class(beam_events_data_t), intent(in) :: data
          logical :: flag
          flag = .true.
        end function beam_events_data_is_generator
```

The number of parameters is two. They are free parameters.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
        procedure :: get_n_par => beam_events_data_get_n_par
```

⟨*SF beam events: procedures*⟩+≡
```
        function beam_events_data_get_n_par (data) result (n)
          class(beam_events_data_t), intent(in) :: data
          integer :: n
          n = 2
        end function beam_events_data_get_n_par
```

⟨*SF beam events: beam events data: TBP*⟩+≡
```
        procedure :: get_pdg_out => beam_events_data_get_pdg_out
```

⟨*SF beam events: procedures*⟩+≡
```
        subroutine beam_events_data_get_pdg_out (data, pdg_out)
          class(beam_events_data_t), intent(in) :: data
          type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
          integer :: i, n
          n = 2
          do i = 1, n
             pdg_out(i) = flavor_get_pdg (data%flv_in(i))
          end do
        end subroutine beam_events_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
        procedure :: allocate_sf_int => beam_events_data_allocate_sf_int
```

⟨*SF beam events: procedures*⟩+≡
```
        subroutine beam_events_data_allocate_sf_int (data, sf_int)
          class(beam_events_data_t), intent(in) :: data
          class(sf_int_t), intent(inout), allocatable :: sf_int
          allocate (beam_events_t :: sf_int)
        end subroutine beam_events_data_allocate_sf_int
```

Output

⟨SF beam events: beam events data: TBP⟩+≡
```
procedure :: write => beam_events_data_write
```

⟨SF beam events: procedures⟩+≡
```
subroutine beam_events_data_write (data, unit, verbose)
  class(beam_events_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "Beam-event file data:"
  write (u, "(3x,A,A,A,A)") "prt_in = ", &
       char (flavor_get_name (data%flv_in(1))), &
       ", ", char (flavor_get_name (data%flv_in(2)))
  write (u, "(3x,A,A,A)") "file  = '", char (data%file), "'"
  write (u, "(3x,A,I0)")  "unit  = ", data%unit
  write (u, "(3x,A,L1)")  "warn  = ", data%warn_eof
end subroutine beam_events_data_write
```

The data file needs to be opened and closed explicitly:

⟨SF beam events: beam events data: TBP⟩+≡
```
procedure :: open => beam_events_data_open
procedure :: close => beam_events_data_close
```

⟨SF beam events: procedures⟩+≡
```
subroutine beam_events_data_open (data)
  class(beam_events_data_t), intent(inout) :: data
  type(string_t) :: filename
  logical :: exist
  if (data%unit == 0) then
     filename = data%file
     if (filename == "") &
         call msg_fatal ("Beam events: $beam_events_file is not set")
     inquire (file = char (filename), exist = exist)
     if (exist) then
        call msg_message ("Beam events: reading from file '" &
             // char (data%file) // "'")
        data%unit = free_unit ()
        open (unit = data%unit, file = char (filename), action = "read", &
             status = "old")
     else
        filename = data%dir // "/" // data%file
        inquire (file = char (filename), exist = exist)
        if (exist) then
           call msg_message ("Beam events: reading from file '" &
                // char (data%file) // "'")
           data%unit = free_unit ()
           open (unit = data%unit, file = char (filename), action = "read", &
                status = "old")
        else
           call msg_fatal ("Beam events: file '" &
                // char (data%file) // "' not found")
        end if
```

```
      end if
   else
      call msg_bug ("Beam events: file '" &
        // char (data%file) // "' is already open")
   end if
end subroutine beam_events_data_open

subroutine beam_events_data_close (data)
  class(beam_events_data_t), intent(inout) :: data
  if (data%unit /= 0) then
     close (data%unit)
     call msg_message ("Beam events: closed file '" &
       // char (data%file) // "'")
     data%unit = 0
  end if
end subroutine beam_events_data_close
```

10.10.2   The beam events object

Flavor and polarization carried through, no radiated particles.

⟨*SF beam events: types*⟩+≡
```
    type, extends (sf_int_t) :: beam_events_t
      type(beam_events_data_t), pointer :: data => null ()
      integer :: count = 0
    contains
    ⟨SF beam events: beam events: TBP⟩
    end type beam_events_t
```

Type string: show beam events file.

⟨*SF beam events: beam events: TBP*⟩≡
```
    procedure :: type_string => beam_events_type_string
```

⟨*SF beam events: procedures*⟩+≡
```
    function beam_events_type_string (object) result (string)
      class(beam_events_t), intent(in) :: object
      type(string_t) :: string
      if (associated (object%data)) then
         string = "Beam events: " // object%data%file
      else
         string = "Beam events: [undefined]"
      end if
    end function beam_events_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF beam events: beam events: TBP*⟩+≡
```
    procedure :: write => beam_events_write
```

⟨*SF beam events: procedures*⟩+≡
```
    subroutine beam_events_write (object, unit, testflag)
      class(beam_events_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
```

1298

```
      integer :: u
      u = output_unit (unit)
      if (associated (object%data)) then
         call object%data%write (u)
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "Beam events data: [undefined]"
      end if
   end subroutine beam_events_write
```

⟨*SF beam events: beam events: TBP*⟩+≡
```
   procedure :: init => beam_events_init
```

⟨*SF beam events: procedures*⟩+≡
```
   subroutine beam_events_init (sf_int, data)
     class(beam_events_t), intent(out) :: sf_int
     class(sf_data_t), intent(in), target :: data
     real(default), dimension(2) :: m2
     real(default), dimension(0) :: mr2
     type(quantum_numbers_mask_t), dimension(4) :: mask
     integer, dimension(4) :: hel_lock
     type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
     type(polarization_t) :: pol1, pol2
     type(state_iterator_t) :: it_hel1, it_hel2
     integer :: i
     select type (data)
     type is (beam_events_data_t)
       m2 = flavor_get_mass (data%flv_in) ** 2
       hel_lock = [3, 4, 1, 2]
       mask = new_quantum_numbers_mask (.false., .false., .false.)
       call sf_int%base_init (mask, m2, mr2, m2, hel_lock = hel_lock)
       sf_int%data => data
       do i = 1, 2
          call quantum_numbers_init (qn_fc(i), &
               flv = data%flv_in(i), &
               col = color_from_flavor (data%flv_in(i)))
          call quantum_numbers_init (qn_fc(i+2), &
               flv = data%flv_in(i), &
               col = color_from_flavor (data%flv_in(i)))
       end do
       call polarization_init_generic (pol1, data%flv_in(1))
       call state_iterator_init (it_hel1, pol1%state)
       do while (state_iterator_is_valid (it_hel1))
          qn_hel(1:1) = state_iterator_get_quantum_numbers (it_hel1)
          qn_hel(3:3) = state_iterator_get_quantum_numbers (it_hel1)
          call polarization_init_generic (pol2, data%flv_in(2))
          call state_iterator_init (it_hel2, pol2%state)
          do while (state_iterator_is_valid (it_hel2))
             qn_hel(2:2) = state_iterator_get_quantum_numbers (it_hel2)
             qn_hel(4:4) = state_iterator_get_quantum_numbers (it_hel2)
             qn = qn_hel .merge. qn_fc
             call interaction_add_state (sf_int%interaction_t, qn)
             call state_iterator_advance (it_hel2)
          end do
```

1299

```
              call polarization_final (pol2)
              call state_iterator_advance (it_hel1)
           end do
           call polarization_final (pol2)
           call interaction_freeze (sf_int%interaction_t)
           call sf_int%set_incoming ([1,2])
           call sf_int%set_outgoing ([3,4])
           call sf_int%data%open ()
           sf_int%status = SF_INITIAL
        end select
     end subroutine beam_events_init
```

This spectrum type needs a finalizer, which closes the data file.

⟨*SF beam events: beam events: TBP*⟩+≡
```
     procedure :: final => sf_beam_events_final
```

⟨*SF beam events: procedures*⟩+≡
```
     subroutine sf_beam_events_final (object)
       class(beam_events_t), intent(inout) :: object
       call object%data%close ()
       call interaction_final (object%interaction_t)
     end subroutine sf_beam_events_final
```

10.10.3 Kinematics

Refer to the `data` component.

⟨*SF beam events: beam events: TBP*⟩+≡
```
     procedure :: is_generator => beam_events_is_generator
```

⟨*SF beam events: procedures*⟩+≡
```
     function beam_events_is_generator (sf_int) result (flag)
       class(beam_events_t), intent(in) :: sf_int
       logical :: flag
       flag = sf_int%data%is_generator ()
     end function beam_events_is_generator
```

Generate free parameters. We read them from file.

⟨*SF beam events: beam events: TBP*⟩+≡
```
     procedure :: generate_free => beam_events_generate_free
```

⟨*SF beam events: procedures*⟩+≡
```
     recursive subroutine beam_events_generate_free (sf_int, r, rb,  x_free)
       class(beam_events_t), intent(inout) :: sf_int
       real(default), dimension(:), intent(out) :: r, rb
       real(default), intent(inout) :: x_free
       integer :: iostat
       associate (data => sf_int%data)
         if (data%unit /= 0) then
            read (data%unit, fmt=*, iostat=iostat)  r
            if (iostat > 0) then
               write (msg_buffer, "(A,I0,A)") &
                    "Beam events: I/O error after reading ", sf_int%count, &
```

```
                    " events"
                call msg_fatal ()
            else if (iostat < 0) then
                if (sf_int%count == 0) then
                    call msg_fatal ("Beam events: file is empty")
                else if (sf_int%data%warn_eof) then
                    write (msg_buffer, "(A,I0,A)") &
                        "Beam events: End of file after reading ", sf_int%count, &
                        " events, rewinding"
                    call msg_warning ()
                end if
                rewind (data%unit)
                sf_int%count = 0
                call sf_int%generate_free (r, rb, x_free)
            else
                sf_int%count = sf_int%count + 1
                rb = 1 - r
                x_free = x_free * product (r)
            end if
        else
            call msg_bug ("Beam events: file is not open for reading")
        end if
    end associate
  end subroutine beam_events_generate_free
```

Set kinematics. Trivial transfer since this is a pure generator.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF beam events: beam events: TBP*⟩+≡
```
    procedure :: complete_kinematics => beam_events_complete_kinematics
```

⟨*SF beam events: procedures*⟩+≡
```
    subroutine beam_events_complete_kinematics (sf_int, x, f, r, rb, map)
      class(beam_events_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(out) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(in) :: r
      real(default), dimension(:), intent(in) :: rb
      logical, intent(in) :: map
      if (map) then
          call msg_fatal ("Beam events: map flag not supported")
      else
          x = r
          f = 1
      end if
      call sf_int%reduce_momenta (x)
    end subroutine beam_events_complete_kinematics
```

Compute inverse kinematics. Trivial in this case.

⟨*SF beam events: beam events: TBP*⟩+≡
```
    procedure :: inverse_kinematics => beam_events_inverse_kinematics
```

⟨*SF beam events: procedures*⟩+≡
```
    subroutine beam_events_inverse_kinematics &
```

```
     (sf_int, x, f, r, rb, map, set_momenta)
   class(beam_events_t), intent(inout) :: sf_int
   real(default), dimension(:), intent(in) :: x
   real(default), intent(out) :: f
   real(default), dimension(:), intent(out) :: r
   real(default), dimension(:), intent(out) :: rb
   logical, intent(in) :: map
   logical, intent(in), optional :: set_momenta
   logical :: set_mom
   set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
   if (map) then
      call msg_fatal ("Beam events: map flag not supported")
   else
      r = x
      f = 1
   end if
   rb = 1 - r
   if (set_mom) then
      call sf_int%reduce_momenta (x)
   end if
end subroutine beam_events_inverse_kinematics
```

10.10.4 Beam events application

Trivial, just set the unit weight.

⟨*SF beam events: beam events: TBP*⟩+≡
```
procedure :: apply => beam_events_apply
```

⟨*SF beam events: procedures*⟩+≡
```
subroutine beam_events_apply (sf_int, scale)
  class(beam_events_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: f
  f = 1
  call interaction_set_matrix_element (sf_int%interaction_t, &
       cmplx (f, kind=default))
  sf_int%status = SF_EVALUATED
end subroutine beam_events_apply
```

10.10.5 Unit tests

⟨*SF beam events: public*⟩+≡
```
public :: sf_beam_events_test
```

⟨*SF beam events: tests*⟩≡
```
subroutine sf_beam_events_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
  ⟨*SF beam events: execute tests*⟩
```
end subroutine sf_beam_events_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF beam events: execute tests*⟩≡
```
call test (sf_beam_events_1, "sf_beam_events_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF beam events: tests*⟩+≡
```
subroutine sf_beam_events_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(pdg_array_t), dimension(2) :: pdg_in
  type(pdg_array_t), dimension(2) :: pdg_out
  integer, dimension(:), allocatable :: pdg1, pdg2
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_beam_events_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &beam-events structure function data"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("QED"), &
       var_str ("QED.mdl"), os_data, model)
  pdg_in(1) = ELECTRON
  pdg_in(2) = -ELECTRON

  allocate (beam_events_data_t :: data)
  select type (data)
  type is (beam_events_data_t)
     call data%init (model, pdg_in, var_str (""), var_str ("beam_events.dat"))
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle codes:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  pdg2 = pdg_out(2)
  write (u, "(2x,99(1x,I0))")  pdg1, pdg2

  call model_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_beam_events_1"

end subroutine sf_beam_events_1
```

1303

g@

Probe the structure-function object

Active the beam event reader, generate an event.

⟨*SF beam events: execute tests*⟩+≡
```
    call test (sf_beam_events_2, "sf_beam_events_2", &
        "generate event", &
        u, results)
```

⟨*SF beam events: tests*⟩+≡
```
    subroutine sf_beam_events_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t), dimension(2) :: flv
      type(pdg_array_t), dimension(2) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k1, k2
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: x_free, f
      integer :: i

      write (u, "(A)")  "* Test output: sf_beam_events_2"
      write (u, "(A)")  "*   Purpose: initialize and display &
          &beam-events structure function data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QED"), &
          var_str ("QED.mdl"), os_data, model)
      call flavor_init (flv(1), ELECTRON, model)
      call flavor_init (flv(2), -ELECTRON, model)
      pdg_in(1) = ELECTRON
      pdg_in(2) = -ELECTRON

      call reset_interaction_counter ()

      allocate (beam_events_data_t :: data)
      select type (data)
      type is (beam_events_data_t)
         call data%init (model, pdg_in, &
             var_str (""), var_str ("test_beam_events.dat"))
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
```

```
call sf_int%set_beam_index ([1,2])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv(1))**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv(2))**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Set dummy parameters and generate x."
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r  = 0
rb = 0
x_free = 1
call sf_int%generate_free (r, rb, x_free)
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free
select type (sf_int)
type is (beam_events_t)
   write (u, "(A,1x,I0)")  "count =", sf_int%count
end select

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
write (u, "(A)")

call sf_int%apply (scale = 0._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate more events, rewind"
write (u, "(A)")

select type (sf_int)
type is (beam_events_t)
   do i = 1, 3
      call sf_int%generate_free (r, rb, x_free)
      write (u, "(A,9(1x,F10.7))")  "r =", r
      write (u, "(A,1x,I0)")  "count =", sf_int%count
   end do
end select
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_beam_events_2"

  end subroutine sf_beam_events_2
```

10.11 LHAPDF

Parton distribution functions (PDFs) are available via an interface to the LHAPDF standard library.

The default PDF for protons set is chosen to be CTEQ6ll (LO fit with LO α_s).

⟨*Limits: public parameters*⟩+≡
```
    character(*), parameter, public :: LHAPDF_DEFAULT_PROTON = "cteq6ll.LHpdf"
    character(*), parameter, public :: LHAPDF_DEFAULT_PION   = "ABFKWPI.LHgrid"
    character(*), parameter, public :: LHAPDF_DEFAULT_PHOTON = "GSG960.LHgrid"
```

10.11.1 The module

⟨`sf_lhapdf.f90`⟩≡
  ⟨*File header*⟩

```
  module sf_lhapdf
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use system_dependencies, only: LHAPDF_PDFSETS_PATH !NODEP!
    use system_dependencies, only: LHAPDF_AVAILABLE !NODEP!
    use limits, only: LHAPDF_DEFAULT_PROTON !NODEP!
    use limits, only: LHAPDF_DEFAULT_PION !NODEP!
    use limits, only: LHAPDF_DEFAULT_PHOTON !NODEP!
    use limits, only: FMT_17, FMT_19 !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use sm_qcd
    use pdg_arrays
    use models
    use flavors
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base
    use hoppet_interface
```

  ⟨*Standard module head*⟩

  ⟨*SF lhapdf: public*⟩

  ⟨*SF lhapdf: types*⟩

  ⟨*SF lhapdf: interfaces*⟩

```
  contains
```

⟨*SF lhapdf: procedures*⟩

⟨*SF lhapdf: tests*⟩

```
end module sf_lhapdf
```

10.11.2 LHAPDF library interface

Here we specify explicit interfaces for all LHAPDF routines that we use below.

⟨*SF lhapdf: interfaces*⟩≡
```
    interface
       subroutine InitPDFsetM (set, file)
         integer, intent(in) :: set
         character(*), intent(in) :: file
       end subroutine InitPDFsetM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine InitPDFM (set, mem)
         integer, intent(in) :: set, mem
       end subroutine InitPDFM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine numberPDFM (set, n_members)
         integer, intent(in) :: set
         integer, intent(out) :: n_members
       end subroutine numberPDFM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine evolvePDFM (set, x, q, ff)
         integer, intent(in) :: set
         double precision, intent(in) :: x, q
         double precision, dimension(-6:6), intent(out) :: ff
       end subroutine evolvePDFM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine evolvePDFphotonM (set, x, q, ff, fphot)
         integer, intent(in) :: set
         double precision, intent(in) :: x, q
         double precision, dimension(-6:6), intent(out) :: ff
         double precision, intent(out) :: fphot
       end subroutine evolvePDFphotonM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine evolvePDFpM (set, x, q, s, scheme, ff)
         integer, intent(in) :: set
         double precision, intent(in) :: x, q, s
         integer, intent(in) :: scheme
         double precision, dimension(-6:6), intent(out) :: ff
       end subroutine evolvePDFpM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine GetXminM (set, mem, xmin)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: xmin
       end subroutine GetXminM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine GetXmaxM (set, mem, xmax)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: xmax
       end subroutine GetXmaxM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine GetQ2minM (set, mem, q2min)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: q2min
       end subroutine GetQ2minM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       subroutine GetQ2maxM (set, mem, q2max)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: q2max
       end subroutine GetQ2maxM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       function has_photon () result(flag)
          logical :: flag
       end function has_photon
    end interface
```

10.11.3 The LHAPDF status

This type holds the initialization status of the LHAPDF system. Entry 1 is for proton PDFs, entry 2 for pion PDFs, entry 3 for photon PDFs.

⟨*SF lhapdf: public*⟩≡
```
public :: lhapdf_status_t
```

⟨*SF lhapdf: types*⟩≡
```
type :: lhapdf_status_t
   private
   logical, dimension(3) :: initialized = .false.
end type lhapdf_status_t
```

⟨*SF lhapdf: public*⟩+≡
```
public :: lhapdf_status_reset
```

⟨*SF lhapdf: procedures*⟩≡
```
subroutine lhapdf_status_reset (lhapdf_status)
  type(lhapdf_status_t), intent(inout) :: lhapdf_status
  lhapdf_status%initialized = .false.
end subroutine lhapdf_status_reset
```

⟨*SF lhapdf: procedures*⟩+≡
```
function lhapdf_status_is_initialized (lhapdf_status, set) result (flag)
  logical :: flag
  type(lhapdf_status_t), intent(in) :: lhapdf_status
  integer, intent(in), optional :: set
  if (present (set)) then
     select case (set)
     case (1:3);    flag = lhapdf_status%initialized(set)
     case default;  flag = .false.
     end select
  else
     flag = any (lhapdf_status%initialized)
  end if
end function lhapdf_status_is_initialized
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_status_set_initialized (lhapdf_status, set)
  type(lhapdf_status_t), intent(inout) :: lhapdf_status
  integer, intent(in) :: set
  lhapdf_status%initialized(set) = .true.
end subroutine lhapdf_status_set_initialized
```

10.11.4 LHAPDF initialization

Before using LHAPDF, we have to initialize it with a particular data set and member. This applies not just if we use structure functions, but also if we just use an α_s formula. The integer `set` should be 1 for proton, 2 for pion, and 3 for photon, but this is just convention.

If the particular set has already been initialized, do nothing. This implies that whenever we want to change the setup for a particular set, we have to

reset the LHAPDF status. `lhapdf_initialize` has an obvious name clash
with `lhapdf_init`, the reason it works for `pdf_builtin` is that there things are
outsourced to a separate module (inc. `lhapdf_status` etc.).

⟨*SF lhapdf: public*⟩+≡
```
    public :: lhapdf_initialize
```

⟨*SF lhapdf: procedures*⟩+≡
```
    subroutine lhapdf_initialize (status, set, prefix, file, member, b_match)
      type(lhapdf_status_t), intent(inout) :: status
      integer, intent(in) :: set
      type(string_t), intent(inout) :: prefix
      type(string_t), intent(inout) :: file
      integer, intent(inout) :: member
      logical, intent(in), optional :: b_match
      if (lhapdf_status_is_initialized (status, set))  return
      if (prefix == "")  prefix = LHAPDF_PDFSETS_PATH
      if (file == "") then
         select case (set)
         case (1);  file = LHAPDF_DEFAULT_PROTON
         case (2);  file = LHAPDF_DEFAULT_PION
         case (3);  file = LHAPDF_DEFAULT_PHOTON
         end select
      end if
      if (data_file_exists (prefix // "/" // file)) then
         call InitPDFsetM (set, char (prefix // "/" // file))
      else
         call msg_fatal ("LHAPDF: Data file '" &
              // char (file) // "' not found in '" // char (prefix) // "'.")
         return
      end if
      if (.not. dataset_member_exists (set, member)) then
         call msg_error (" LHAPDF: Chosen member does not exist for set '" &
              // char (file) // "', using default.")
         member = 0
      end if
      call InitPDFM (set, member)
      if (present (b_match)) then
         if (b_match)  call hoppet_init ()
      end if
      call lhapdf_status_set_initialized (status, set)
    contains
      function data_file_exists (fq_name) result (exist)
        type(string_t), intent(in) :: fq_name
        logical :: exist
        inquire (file = char(fq_name), exist = exist)
      end function data_file_exists
      function dataset_member_exists (set, member) result (exist)
        integer, intent(in) :: set, member
        logical :: exist
        integer :: n_members
        call numberPDFM (set, n_members)
        exist = member >= 0 .and. member <= n_members
      end function dataset_member_exists
    end subroutine lhapdf_initialize
```

10.11.5 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF lhapdf: lhapdf: TBP*⟩≡
```
procedure :: complete_kinematics => lhapdf_complete_kinematics
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_complete_kinematics (sf_int, x, f, r, rb, map)
  class(lhapdf_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default) :: xb1
  if (map) then
     call msg_fatal ("LHAPDF: map flag not supported")
  else
     x(1) = r(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)
     sf_int%x = x(1)
  case (SF_FAILED_KINEMATICS)
     sf_int%x = 0
     f = 0
  end select
end subroutine lhapdf_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
procedure :: inverse_kinematics => lhapdf_inverse_kinematics
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(lhapdf_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: xb1
```

```
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      if (map) then
         call msg_fatal ("LHAPDF: map flag not supported")
      else
         r(1) = x(1)
         f = 1
      end if
      xb1 = 1 - x(1)
      rb = 1 - r
      if (set_mom) then
         call sf_int%split_momentum (x, xb1)
         select case (sf_int%status)
         case (SF_DONE_KINEMATICS)
            sf_int%x = x(1)
         case (SF_FAILED_KINEMATICS)
            sf_int%x = 0
            f = 0
         end select
      end if
    end subroutine lhapdf_inverse_kinematics
```

10.11.6   The LHAPDF data block

The data block holds the incoming flavor (which has to be proton, pion, or photon), the corresponding pointer to the global access data (1, 2, or 3), the flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF for the specified set, and a mask that determines which partons will be actually in use.

⟨*SF lhapdf: public*⟩+≡
```
    public :: lhapdf_data_t
```

⟨*SF lhapdf: types*⟩+≡
```
    type, extends (sf_data_t) :: lhapdf_data_t
      private
      type(string_t) :: prefix
      type(string_t) :: file
      integer :: member = 0
      type(model_t), pointer :: model => null ()
      type(flavor_t) :: flv_in
      integer :: set = 0
      logical :: invert = .false.
      logical :: photon = .false.
      logical :: has_photon = .false.
      integer :: photon_scheme = 0
      real(default) :: xmin = 0, xmax = 0
      real(default) :: qmin = 0, qmax = 0
      logical, dimension(-6:6) :: mask = .true.
      logical :: mask_photon = .true.
      logical :: hoppet_b_matching = .false.
    contains
      ⟨SF lhapdf: lhapdf data: TBP⟩
```

```
      end type lhapdf_data_t
```

Generate PDF data. This is a provided as a function, but it has the side-effect of initializing the requested PDF set. A finalizer is not needed.

The library uses double precision, so since the default precision may be quadruple, we use auxiliary variables for type casting.

⟨*SF lhapdf: lhapdf data: TBP*⟩≡
```
      procedure :: init => lhapdf_data_init
```

⟨*SF lhapdf: procedures*⟩+≡
```
      subroutine lhapdf_data_init &
          (data, status, model, pdg_in, prefix, file, member, photon_scheme, &
              hoppet_b_matching)
        class(lhapdf_data_t), intent(out) :: data
        type(lhapdf_status_t), intent(inout) :: status
        type(model_t), intent(in), target :: model
        type(pdg_array_t), intent(in) :: pdg_in
        type(string_t), intent(in), optional :: prefix, file
        integer, intent(in), optional :: member
        integer, intent(in), optional :: photon_scheme
        logical, intent(in), optional :: hoppet_b_matching
        double precision :: xmin, xmax, q2min, q2max
        external :: InitPDFsetM, InitPDFM, numberPDFM
        external :: GetXminM, GetXmaxM, GetQ2minM, GetQ2maxM
        if (.not. LHAPDF_AVAILABLE) then
           call msg_fatal ("LHAPDF requested but library is not linked")
           return
        end if
        data%model => model
        if (pdg_array_get_length (pdg_in) /= 1) &
             call msg_fatal ("PDF: incoming particle must be unique")
        call flavor_init (data%flv_in, pdg_array_get (pdg_in, 1), model)
        select case (pdg_array_get (pdg_in, 1))
        case (PROTON)
           data%set = 1
        case (-PROTON)
           data%set = 1
           data%invert = .true.
        case (PIPLUS)
           data%set = 2
        case (-PIPLUS)
           data%set = 2
           data%invert = .true.
        case (PHOTON)
           data%set = 3
           data%photon = .true.
           if (present (photon_scheme))  data%photon_scheme = photon_scheme
        case default
           call msg_fatal (" LHAPDF: " &
                // "incoming particle must be (anti)proton, pion, or photon.")
           return
        end select
        if (present (prefix)) then
           data%prefix = prefix
```

1314

```
        else
           data%prefix = ""
        end if
        if (present (file)) then
           data%file = file
        else
           data%file = ""
        end if
        if (present (hoppet_b_matching))  data%hoppet_b_matching = hoppet_b_matching
        call lhapdf_initialize &
             (status, data%set, data%prefix, data%file, data%member, &
             data%hoppet_b_matching)
        call GetXminM (data%set, data%member, xmin)
        call GetXmaxM (data%set, data%member, xmax)
        call GetQ2minM (data%set, data%member, q2min)
        call GetQ2maxM (data%set, data%member, q2max)
        data%xmin = xmin
        data%xmax = xmax
        data%qmin = sqrt (q2min)
        data%qmax = sqrt (q2max)
        data%has_photon = has_photon ()
      end subroutine lhapdf_data_init
```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF
will generate the corresponding flavor on output.

⟨*LHAPDF: lhapdf data: TBP*⟩≡
```
    procedure :: set_mask => lhapdf_data_set_mask
```

⟨*LHAPDF: procedures*⟩≡
```
    subroutine lhapdf_data_set_mask (data, mask)
      class(lhapdf_data_t), intent(inout) :: data
      logical, dimension(-6:6), intent(in) :: mask
      data%mask = mask
    end subroutine lhapdf_data_set_mask
```

Return the public part of the data set.

⟨*LHAPDF: public*⟩≡
```
    public :: lhapdf_data_get_public_info
```

⟨*LHAPDF: procedures*⟩+≡
```
    subroutine lhapdf_data_get_public_info &
         (data, lhapdf_dir, lhapdf_file, lhapdf_member)
      type(lhapdf_data_t), intent(in) :: data
      type(string_t), intent(out) :: lhapdf_dir, lhapdf_file
      integer, intent(out) :: lhapdf_member
      lhapdf_dir = data%prefix
      lhapdf_file = data%file
      lhapdf_member = data%member
    end subroutine lhapdf_data_get_public_info
```

Return the number of the member of the data set.

⟨*LHAPDF: public*⟩+≡
```
    public :: lhapdf_data_get_set
```

⟨*LHAPDF: procedures*⟩+≡
```
function lhapdf_data_get_set(data) result(set)
  type(lhapdf_data_t), intent(in) :: data
  integer :: set
  set = data%set
end function lhapdf_data_get_set
```

Output

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
procedure :: write => lhapdf_data_write
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_data_write (data, unit, verbose)
  class(lhapdf_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: u
  if (present (verbose)) then
     verb = verbose
  else
     verb = .false.
  end if
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "LHAPDF data:"
  if (data%set /= 0) then
     write (u, "(3x,A)", advance="no") "flavor      = "
     call flavor_write (data%flv_in, u);  write (u, *)
     if (verb) then
        write (u, "(3x,A,A)")        " prefix      = ", char (data%prefix)
     else
        write (u, "(3x,A,A)")        " prefix      = ", &
             " <empty (non-verbose version)>"
     end if
     write (u, "(3x,A,A)")        " file        = ", char (data%file)
     write (u, "(3x,A,I3)")       " member      = ", data%member
     write (u, "(3x,A," // FMT_19 // ")") " x(min)      = ", data%xmin
     write (u, "(3x,A," // FMT_19 // ")") " x(max)      = ", data%xmax
     write (u, "(3x,A," // FMT_19 // ")") " Q^2(min)    = ", data%qmin
     write (u, "(3x,A," // FMT_19 // ")") " Q^2(max)    = ", data%qmax
     write (u, "(3x,A,L1)")       " invert      = ", data%invert
     if (data%photon)  write (u, "(3x,A,I3)") &
          " IP2 (scheme) = ", data%photon_scheme
        write (u, "(3x,A,6(1x,L1),1x,A,1x,L1,1x,A,6(1x,L1))") &
             " mask        = ", &
             data%mask(-6:-1), "*", data%mask(0), "*", data%mask(1:6)
        write (u, "(3x,A,L1)")   " photon mask  = ", data%mask_photon
     if (data%set == 1)  write (u, "(3x,A,L1)") &
          " hoppet_b     = ", data%hoppet_b_matching
  else
     write (u, "(3x,A)") "[undefined]"
  end if
end subroutine lhapdf_data_write
```

The number of parameters is one. We do not generate transverse momentum.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
    procedure :: get_n_par => lhapdf_data_get_n_par
```

⟨*SF lhapdf: procedures*⟩+≡
```
    function lhapdf_data_get_n_par (data) result (n)
      class(lhapdf_data_t), intent(in) :: data
      integer :: n
      n = 1
    end function lhapdf_data_get_n_par
```

Return the outgoing particle PDG codes. This is based on the mask.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
    procedure :: get_pdg_out => lhapdf_data_get_pdg_out
```

⟨*SF lhapdf: procedures*⟩+≡
```
    subroutine lhapdf_data_get_pdg_out (data, pdg_out)
      class(lhapdf_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      integer, dimension(:), allocatable :: pdg1
      integer :: n, np, i
      n = count (data%mask)
      np = 0;  if (data%has_photon .and. data%mask_photon)  np = 1
      allocate (pdg1 (n + np))
      pdg1(1:n) = pack ([(i, i = -6, 6)], data%mask)
      if (np == 1)  pdg1(n+np) = PHOTON
      pdg_out(1) = pdg1
    end subroutine lhapdf_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
    procedure :: allocate_sf_int => lhapdf_data_allocate_sf_int
```

⟨*SF lhapdf: procedures*⟩+≡
```
    subroutine lhapdf_data_allocate_sf_int (data, sf_int)
      class(lhapdf_data_t), intent(in) :: data
      class(sf_int_t), intent(inout), allocatable :: sf_int
      allocate (lhapdf_t :: sf_int)
    end subroutine lhapdf_data_allocate_sf_int
```

Return the numerical PDF set index.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
    procedure :: get_pdf_set => lhapdf_data_get_pdf_set
```

⟨*SF lhapdf: procedures*⟩+≡
```
    function lhapdf_data_get_pdf_set (data) result (pdf_set)
      class(lhapdf_data_t), intent(in) :: data
      integer :: pdf_set
      pdf_set = data%set
    end function lhapdf_data_get_pdf_set
```

10.11.7 The LHAPDF object

The `lhapdf_t` data type is a $1 \to 2$ interaction which describes the splitting of an (anti)proton into a parton and a beam remnant. We stay in the strict forward-splitting limit, but allow some invariant mass for the beam remnant such that the outgoing parton is exactly massless. For a real event, we would replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

This is the LHAPDF object which holds input data together with the interaction. We also store the x momentum fraction and the scale, since kinematics and function value are requested at different times.

The PDF application is a $1 \to 2$ splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

⟨*SF lhapdf: types*⟩+≡
```
type, extends (sf_int_t) :: lhapdf_t
  type(lhapdf_data_t), pointer :: data => null ()
  real(default) :: x = 0
  real(default) :: q = 0
  real(default) :: s = 0
 contains
 ⟨SF lhapdf: lhapdf: TBP⟩
 end type lhapdf_t
```

Type string: display the chosen PDF set.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
procedure :: type_string => lhapdf_type_string
```

⟨*SF lhapdf: procedures*⟩+≡
```
function lhapdf_type_string (object) result (string)
  class(lhapdf_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "LHAPDF: " // object%data%file
  else
     string = "LHAPDF: [undefined]"
  end if
end function lhapdf_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
procedure :: write => lhapdf_write
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_write (object, unit, testflag)
  class(lhapdf_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
```

```
         if (object%status >= SF_DONE_KINEMATICS) then
            write (u, "(1x,A)")  "SF parameters:"
            write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
            if (object%status >= SF_FAILED_EVALUATION) then
               write (u, "(3x,A," // FMT_17 // ")")  "Q =", object%q
            end if
         end if
         call object%base_write (u, testflag)
      else
         write (u, "(1x,A)")  "LHAPDF data: [undefined]"
      end if
   end subroutine lhapdf_write
```

Initialize. We know that `data` will be of concrete type `sf_lhapdf_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
      procedure :: init => lhapdf_init
```

⟨*SF lhapdf: procedures*⟩+≡
```
   subroutine lhapdf_init (sf_int, data)
     class(lhapdf_t), intent(out) :: sf_int
     class(sf_data_t), intent(in), target :: data
     type(quantum_numbers_mask_t), dimension(3) :: mask
     type(flavor_t) :: flv, flv_remnant
     type(quantum_numbers_t), dimension(3) :: qn
     integer :: i
     select type (data)
     type is (lhapdf_data_t)
        mask = new_quantum_numbers_mask (.false., .false., .true.)
        call sf_int%base_init (mask, [0._default], [0._default], [0._default])
        sf_int%data => data
        do i = -6, 6
           if (data%mask(i)) then
              call quantum_numbers_init (qn(1), data%flv_in)
              if (i == 0) then
                 call flavor_init (flv, GLUON, data%model)
                 call flavor_init (flv_remnant, HADRON_REMNANT_OCTET, data%model)
              else
                 call flavor_init (flv, i, data%model)
                 call flavor_init (flv_remnant, &
                       sign (HADRON_REMNANT_TRIPLET, -i), data%model)
              end if
              call quantum_numbers_init (qn(2), &
                    flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
              call quantum_numbers_init (qn(3), &
                    flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
              call interaction_add_state (sf_int%interaction_t, qn)
           end if
        end do
        if (data%has_photon .and. data%mask_photon) then
           call flavor_init (flv, PHOTON, data%model)
           call flavor_init (flv_remnant, HADRON_REMNANT_SINGLET, data%model)
```

```
              call quantum_numbers_init (qn(2), flv = flv_remnant, &
                   col = color_from_flavor (flv_remnant, 1))
              call quantum_numbers_init (qn(3), flv = flv, &
                   col = color_from_flavor (flv, 1, reverse=.true.))
              call interaction_add_state (sf_int%interaction_t, qn)
         end if
         call interaction_freeze (sf_int%interaction_t)
         call sf_int%set_incoming ([1])
         call sf_int%set_radiated ([2])
         call sf_int%set_outgoing ([3])
         sf_int%status = SF_INITIAL
      end select
    end subroutine lhapdf_init
```

10.11.8 Structure function

We have to cast the LHAPDF arguments to/from double precision (possibly from/to quadruple precision), if necessary. Furthermore, some structure functions can yield negative results (sea quarks close to $x = 1$). We set these unphysical values to zero.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
    procedure :: apply => lhapdf_apply
```

⟨*SF lhapdf: procedures*⟩+≡
```
    subroutine lhapdf_apply (sf_int, scale)
      class(lhapdf_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
      real(default) :: x, s
      double precision :: xx, qq, ss
      double precision, dimension(-6:6) :: ff
      double precision :: fphot
      complex(default), dimension(:), allocatable :: fc
      external :: evolvePDFM, evolvePDFpM
      associate (data => sf_int%data)
        sf_int%q = scale
        x = sf_int%x
        s = sf_int%s
        qq = min (data% qmax, scale)
        qq = max (data% qmin, qq)
        if (.not. data% photon) then
           xx = x
           if (data% invert) then
              if (data%has_photon) then
                 call evolvePDFphotonM (data% set, xx, qq, ff(6:-6:-1), fphot)
              else
                 if (data%hoppet_b_matching) then
                    call hoppet_eval (xx, qq, ff(6:-6:-1))
                 else
                    call evolvePDFM (data% set, xx, qq, ff(6:-6:-1))
                 end if
              end if
           else
              if (data%has_photon) then
```

```
                call evolvePDFphotonM (data% set, xx, qq, ff, fphot)
            else
                if (data%hoppet_b_matching) then
                    call hoppet_eval (xx, qq, ff)
                else
                    call evolvePDFM (data% set, xx, qq, ff)
                end if
            end if
        end if
    else
        ss = s
        call evolvePDFpM (data% set, xx, qq, &
            ss, data% photon_scheme, ff)
    end if
    if (data%has_photon) then
        allocate (fc (count ([data%mask, data%mask_photon])))
        fc = max (pack ([ff, fphot] / x, &
            [data% mask, data%mask_photon]), 0._default)
    else
        allocate (fc (count (data%mask)))
        fc = max (pack (ff / x, data%mask), 0._default)
    end if
    end associate
    call interaction_set_matrix_element (sf_int%interaction_t, fc)
    sf_int%status = SF_EVALUATED
end subroutine lhapdf_apply
```

10.11.9 Strong Coupling

Since the PDF codes provide a function for computing the running α_s value, we make this available as an implementation of the abstract `alpha_qcd_t` type, which is used for matrix element evaluation.

⟨*SF lhapdf: public*⟩+≡
```
    public :: alpha_qcd_lhapdf_t
```

⟨*SF lhapdf: types*⟩+≡
```
    type, extends (alpha_qcd_t) :: alpha_qcd_lhapdf_t
      type(string_t) :: pdfset_dir
      type(string_t) :: pdfset_file
      integer :: pdfset_member = -1
    contains
    ⟨SF lhapdf: alpha qcd: TBP⟩
    end type alpha_qcd_lhapdf_t
```

Output. As in earlier versions we leave the LHAPDF path out.

⟨*SF lhapdf: alpha qcd: TBP*⟩≡
```
    procedure :: write => alpha_qcd_lhapdf_write
```

⟨*SF lhapdf: procedures*⟩+≡
```
    subroutine alpha_qcd_lhapdf_write (object, unit)
      class(alpha_qcd_lhapdf_t), intent(in) :: object
      integer, intent(in), optional :: unit
```

```
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A)")  "QCD parameters (lhapdf):"
    write (u, "(5x,A,A)")  "PDF set    = ", char (object%pdfset_file)
    write (u, "(5x,A,I0)")  "PDF member = ", object%pdfset_member
  end subroutine alpha_qcd_lhapdf_write
```

Calculation: the numeric member ID selects the correct PDF set, which must be properly initialized.

⟨*SF lhapdf: interfaces*⟩+≡
```
    interface
       double precision function alphasPDF (Q)
         double precision, intent(in) :: Q
       end function alphasPDF
    end interface
```

⟨*SF lhapdf: alpha qcd: TBP*⟩+≡
```
    procedure :: get => alpha_qcd_lhapdf_get
```

⟨*SF lhapdf: procedures*⟩+≡
```
    function alpha_qcd_lhapdf_get (alpha_qcd, scale) result (alpha)
      class(alpha_qcd_lhapdf_t), intent(in) :: alpha_qcd
      real(default), intent(in) :: scale
      real(default) :: alpha
      alpha = alphasPDF (dble (scale))
    end function alpha_qcd_lhapdf_get
```

Initialization. We need to access the (quasi-global) initialization status.

⟨*SF lhapdf: alpha qcd: TBP*⟩+≡
```
    procedure :: init => alpha_qcd_lhapdf_init
```

⟨*SF lhapdf: procedures*⟩+≡
```
    subroutine alpha_qcd_lhapdf_init (alpha_qcd, status, file, member, path)
      class(alpha_qcd_lhapdf_t), intent(out) :: alpha_qcd
      type(lhapdf_status_t), intent(inout) :: status
      type(string_t), intent(inout) :: file
      integer, intent(inout) :: member
      type(string_t), intent(inout) :: path
      alpha_qcd%pdfset_file = file
      alpha_qcd%pdfset_member = member
      if (alpha_qcd%pdfset_member < 0) &
           call msg_fatal ("QCD parameter initialization: PDF set " &
           // char (file) // " is unknown")
      call lhapdf_initialize (status, 1, path, file, member)
    end subroutine alpha_qcd_lhapdf_init
```

10.11.10 Unit tests

⟨*SF lhapdf: public*⟩+≡
```
    public :: sf_lhapdf_test
```

⟨*SF lhapdf: tests*⟩≡
```
subroutine sf_lhapdf_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SF lhapdf: execute tests⟩
end subroutine sf_lhapdf_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF lhapdf: execute tests*⟩≡
```
call test (sf_lhapdf_1, "sf_lhapdf_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF lhapdf: tests*⟩+≡
```
subroutine sf_lhapdf_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  class(sf_data_t), allocatable :: data
  type(lhapdf_status_t) :: status

  write (u, "(A)")  "* Test output: sf_lhapdf_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &test structure function data"
  write (u, "(A)")

  write (u, "(A)")  "* Create empty data object"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("QCD"), &
       var_str ("QCD.mdl"), os_data, model)
  pdg_in = PROTON

  allocate (lhapdf_data_t :: data)
  call data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize"
  write (u, "(A)")

  select type (data)
  type is (lhapdf_data_t)
     call data%init (status, model, pdg_in)
  end select
```

1323

```
    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_lhapdf_1"

  end subroutine sf_lhapdf_1
```

Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

⟨SF lhapdf: execute tests⟩+≡
```
    call test (sf_lhapdf_2, "sf_lhapdf_2", &
         "structure function instance", &
         u, results)
```

⟨SF lhapdf: tests⟩+≡
```
    subroutine sf_lhapdf_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(lhapdf_status_t) :: status
      type(vector4_t) :: k
      type(vector4_t), dimension(2) :: q
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_lhapdf_2"
      write (u, "(A)")  "*   Purpose: initialize and fill &
           &test structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
```

```
call model_list%read_model (var_str ("QCD"), &
     var_str ("QCD.mdl"), os_data, model)
call flavor_init (flv, PROTON, model)
pdg_in = PROTON

call reset_interaction_counter ()

allocate (lhapdf_data_t :: data)
select type (data)
type is (lhapdf_data_t)
   call data%init (status, model, pdg_in)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.5_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)
```

```
      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
      call sf_int%set_beam_index ([1])

      call sf_int%seed_kinematics ([k])
      call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
      call sf_int%recover_x (x)

      write (u, "(A,9(1x,F10.7))")  "x =", x

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate for Q = 100 GeV"
      write (u, "(A)")

      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%apply (scale = 100._default)
      call sf_int%write (u, testflag = .true.)


      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_lhapdf_2"

    end subroutine sf_lhapdf_2
```

Strong Coupling

Test α_s as an implementation of the `alpha_qcd_t` abstract type.

⟨*SF lhapdf: execute tests*⟩+≡
```
    call test (sf_lhapdf_3, "sf_lhapdf_3", &
         "running alpha_s", &
         u, results)
```

⟨*SF lhapdf: tests*⟩+≡
```
    subroutine sf_lhapdf_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(lhapdf_status_t) :: status
      type(qcd_t) :: qcd
      type(string_t) :: name, path
      integer :: member

      write (u, "(A)")  "* Test output: sf_lhapdf_3"
      write (u, "(A)")  "*   Purpose: initialize and evaluate alpha_s"
      write (u, "(A)")
```

```
      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)

      name = "cteq6ll.LHpdf"
      member = 1
      path = ""

      write (u, "(A)")  "* Initialize qcd object"
      write (u, "(A)")

      allocate (alpha_qcd_lhapdf_t :: qcd%alpha)
      select type (alpha => qcd%alpha)
      type is (alpha_qcd_lhapdf_t)
         call alpha%init (status, name, member, path)
      end select
      call qcd%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate for Q = 100"
      write (u, "(A)")

      write (u, "(1x,A,F8.5)")  "alpha = ", qcd%alpha%get (100._default)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call model_list%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_lhapdf_3"

    end subroutine sf_lhapdf_3
```

10.12   Builtin PDF sets

For convenience in order not to depend on the external package LHAPDF, we
ship some PDFs with WHIZARD.

⟨*Limits: public parameters*⟩+≡
```
    character(*), parameter, public :: PDF_BUILTIN_DEFAULT_PROTON = "CTEQ6L"
    character(*), parameter, public :: PDF_BUILTIN_DEFAULT_PION   = "NONE"
    character(*), parameter, public :: PDF_BUILTIN_DEFAULT_PHOTON = "MRST2004QEDp"
```

10.12.1   The module

⟨`sf_pdf_builtin.f90`⟩≡
  ⟨*File header*⟩

  module sf_pdf_builtin

⟨*Use kinds*⟩
⟨*Use strings*⟩
```
  use limits, only: PDF_BUILTIN_DEFAULT_PROTON !NODEP!
  use limits, only: PDF_BUILTIN_DEFAULT_PION !NODEP!
  use limits, only: PDF_BUILTIN_DEFAULT_PHOTON !NODEP!
  use limits, only: FMT_17 !NODEP!
```
⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use lorentz !NODEP!
  use pdf_builtin !NODEP!
  use unit_tests
  use os_interface
  use sm_qcd
  use pdg_arrays
  use models
  use flavors
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use interactions
  use sf_aux
  use sf_base
```

⟨*Standard module head*⟩

⟨*SF pdf builtin: public*⟩

⟨*SF pdf builtin: types*⟩

```
contains
```

⟨*SF pdf builtin: procedures*⟩

⟨*SF pdf builtin: tests*⟩

```
end module sf_pdf_builtin
```

10.12.2 The PDF builtin data block

The data block holds the incoming flavor (which has to be proton, pion, or photon), the corresponding pointer to the global access data (1, 2, or 3), the flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF for the specified set, and a mask that determines which partons will be actually in use.

⟨*SF pdf builtin: public*⟩≡
```
    public :: pdf_builtin_data_t
```

⟨*SF pdf builtin: types*⟩≡
```
    type, extends (sf_data_t) :: pdf_builtin_data_t
       private
       integer :: id = -1
```

```
        type (string_t) :: name
        type(model_t), pointer :: model => null ()
        type(flavor_t) :: flv_in
        logical :: invert
        logical :: has_photon
        logical :: photon
        logical, dimension(-6:6) :: mask
        logical :: mask_photon
   contains
   ⟨SF pdf builtin: pdf builtin data: TBP⟩
   end type pdf_builtin_data_t
```

Generate PDF data and initialize the requested set. Pion and photon PDFs are disabled at the moment until we ship appropiate structure functions. needed.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩≡
```
   procedure :: init => pdf_builtin_data_init
```

⟨*SF pdf builtin: procedures*⟩≡
```
   subroutine pdf_builtin_data_init (data, pdf_status, model, pdg_in, name, path)
     class(pdf_builtin_data_t), intent(out) :: data
     type(pdf_builtin_status_t), intent(inout) :: pdf_status
     type(model_t), intent(in), target :: model
     type(pdg_array_t), intent(in) :: pdg_in
     type(string_t), intent(in) :: name
     type(string_t), intent(in) :: path
     data%model => model
     if (pdg_array_get_length (pdg_in) /= 1) &
          call msg_fatal ("PDF: incoming particle must be unique")
     call flavor_init (data%flv_in, pdg_array_get (pdg_in, 1), model)
     data%mask = .true.
     data%mask_photon = .true.
     select case (pdg_array_get (pdg_in, 1))
     case (PROTON)
        data%name = var_str (PDF_BUILTIN_DEFAULT_PROTON)
        data%invert = .false.
        data%photon = .false.
     case (-PROTON)
        data%name = var_str (PDF_BUILTIN_DEFAULT_PROTON)
        data%invert = .true.
        data%photon = .false.
        ! case (PIPLUS)
        !    data%name = var_str (PDF_BUILTIN_DEFAULT_PION)
        !    data%invert = .false.
        !    data%photon = .false.
        ! case (-PIPLUS)
        !    data%name = var_str (PDF_BUILTIN_DEFAULT_PION)
        !    data%invert = .true.
        !    data%photon = .false.
        ! case (PHOTON)
        !    data%name = var_str (PDF_BUILTIN_DEFAULT_PHOTON)
        !    data%invert = .false.
        !    data%photon = .true.
     case default
        call msg_fatal ("PDF: " &
```

```
                // "incoming particle must either proton or antiproton.")
          return
      end select
      data%name = name
      data%id = pdf_get_id (data%name)
      if (data%id < 0) call msg_fatal ("unknown PDF set " // char (data%name))
      data%has_photon = pdf_provides_photon (data%id)
      call pdf_init (pdf_status, data%id, path)
    end subroutine pdf_builtin_data_init
```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF
will generate the corresponding flavor on output.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡
```
    procedure :: set_mask => pdf_builtin_data_set_mask
```

⟨*SF pdf builtin: procedures*⟩+≡
```
    subroutine pdf_builtin_data_set_mask (data, mask)
      class(pdf_builtin_data_t), intent(inout) :: data
      logical, dimension(-6:6), intent(in) :: mask
      data%mask = mask
    end subroutine pdf_builtin_data_set_mask
```

Output.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡
```
    procedure :: write => pdf_builtin_data_write
```

⟨*SF pdf builtin: procedures*⟩+≡
```
    subroutine pdf_builtin_data_write (data, unit, verbose)
      class(pdf_builtin_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)")  "PDF builtin data:"
      if (data%id < 0) then
         write (u, "(3x,A)") "[undefined]"
         return
      end if
      write (u, "(3x,A)", advance="no") "flavor        = "
      call flavor_write (data%flv_in, u);  write (u, *)
      write (u, "(3x,A,A)")   "name          = ", char (data%name)
      write (u, "(3x,A,L1)")  "invert        = ", data%invert
      write (u, "(3x,A,L1)")  "has photon    = ", data%has_photon
      write (u, "(3x,A,6(1x,L1),1x,A,1x,L1,1x,A,6(1x,L1))") &
           "mask          =", &
           data%mask(-6:-1), "*", data%mask(0), "*", data%mask(1:6)
      write (u, "(3x,A,L1)") "photon mask  = ", data%mask_photon
    end subroutine pdf_builtin_data_write
```

The number of parameters is one. We do not generate transverse momentum.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡
```
    procedure :: get_n_par => pdf_builtin_data_get_n_par
```

⟨*SF pdf builtin: procedures*⟩+≡
```
function pdf_builtin_data_get_n_par (data) result (n)
  class(pdf_builtin_data_t), intent(in) :: data
  integer :: n
  n = 1
end function pdf_builtin_data_get_n_par
```

Return the outgoing particle PDG codes. This is based on the mask.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡
```
procedure :: get_pdg_out => pdf_builtin_data_get_pdg_out
```

⟨*SF pdf builtin: procedures*⟩+≡
```
subroutine pdf_builtin_data_get_pdg_out (data, pdg_out)
  class(pdf_builtin_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  integer :: n, np, i
  n = count (data%mask)
  np = 0;  if (data%has_photon .and. data%mask_photon)  np = 1
  allocate (pdg1 (n + np))
  pdg1(1:n) = pack ([(i, i = -6, 6)], data%mask)
  if (np == 1)  pdg1(n+np) = PHOTON
  pdg_out(1) = pdg1
end subroutine pdf_builtin_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡
```
procedure :: allocate_sf_int => pdf_builtin_data_allocate_sf_int
```

⟨*SF pdf builtin: procedures*⟩+≡
```
subroutine pdf_builtin_data_allocate_sf_int (data, sf_int)
  class(pdf_builtin_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (pdf_builtin_t :: sf_int)
end subroutine pdf_builtin_data_allocate_sf_int
```

Return the numerical PDF set index.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡
```
procedure :: get_pdf_set => pdf_builtin_data_get_pdf_set
```

⟨*SF pdf builtin: procedures*⟩+≡
```
function pdf_builtin_data_get_pdf_set (data) result (pdf_set)
  class(pdf_builtin_data_t), intent(in) :: data
  integer :: pdf_set
  pdf_set = data%id
end function pdf_builtin_data_get_pdf_set
```

10.12.3 The PDF object

The PDF $1 \rightarrow 2$ interaction which describes the splitting of an (anti)proton into a parton and a beam remnant. We stay in the strict forward-splitting limit, but allow some invariant mass for the beam remnant such that the outgoing parton

is exactly massless. For a real event, we would replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

The PDF application is a $1 \to 2$ splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

⟨*SF pdf builtin: types*⟩+≡
```
type, extends (sf_int_t) :: pdf_builtin_t
   type(pdf_builtin_data_t), pointer :: data => null ()
   real(default) :: x = 0
   real(default) :: q = 0
 contains
 ⟨SF pdf builtin: pdf builtin: TBP⟩
 end type pdf_builtin_t
```

Type string: display the chosen PDF set.

⟨*SF pdf builtin: pdf builtin: TBP*⟩≡
```
procedure :: type_string => pdf_builtin_type_string
```

⟨*SF pdf builtin: procedures*⟩+≡
```
function pdf_builtin_type_string (object) result (string)
  class(pdf_builtin_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "PDF builtin: " // object%data%name
  else
     string = "PDF builtin: [undefined]"
  end if
end function pdf_builtin_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡
```
procedure :: write => pdf_builtin_write
```

⟨*SF pdf builtin: procedures*⟩+≡
```
subroutine pdf_builtin_write (object, unit, testflag)
  class(pdf_builtin_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
           write (u, "(3x,A," // FMT_17 // ")")  "Q =", object%q
        end if
     end if
     call object%base_write (u, testflag)
  else
```

```
          write (u, "(1x,A)")  "PDF builtin data: [undefined]"
        end if
      end subroutine pdf_builtin_write
```

Initialize. We know that `data` will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡
```
    procedure :: init => pdf_builtin_init
```

⟨*SF pdf builtin: procedures*⟩+≡
```
    subroutine pdf_builtin_init (sf_int, data)
      class(pdf_builtin_t), intent(out) :: sf_int
      class(sf_data_t), intent(in), target :: data
      type(quantum_numbers_mask_t), dimension(3) :: mask
      type(flavor_t) :: flv, flv_remnant
      type(quantum_numbers_t), dimension(3) :: qn
      integer :: i
      select type (data)
      type is (pdf_builtin_data_t)
         mask = new_quantum_numbers_mask (.false., .false., .true.)
         call sf_int%base_init (mask, [0._default], [0._default], [0._default])
         sf_int%data => data
         do i = -6, 6
            if (data%mask(i)) then
               call quantum_numbers_init (qn(1), data%flv_in)
               if (i == 0) then
                  call flavor_init (flv, GLUON, data%model)
                  call flavor_init (flv_remnant, HADRON_REMNANT_OCTET, data%model)
               else
                  call flavor_init (flv, i, data%model)
                  call flavor_init (flv_remnant, &
                       sign (HADRON_REMNANT_TRIPLET, -i), data%model)
               end if
               call quantum_numbers_init (qn(2), &
                    flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
               call quantum_numbers_init (qn(3), &
                    flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
               call interaction_add_state (sf_int%interaction_t, qn)
            end if
         end do
         if (data%has_photon .and. data%mask_photon) then
            call flavor_init (flv, PHOTON, data%model)
            call flavor_init (flv_remnant, HADRON_REMNANT_SINGLET, data%model)
            call quantum_numbers_init (qn(2), flv = flv_remnant, &
                 col = color_from_flavor (flv_remnant, 1))
            call quantum_numbers_init (qn(3), flv = flv, &
                 col = color_from_flavor (flv, 1, reverse = .true.))
            call interaction_add_state (sf_int%interaction_t, qn)
         end if
         call interaction_freeze (sf_int%interaction_t)
```

```
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
      sf_int%status = SF_INITIAL
   end select
 end subroutine pdf_builtin_init
```

10.12.4 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡
```
     procedure :: complete_kinematics => pdf_builtin_complete_kinematics
```

⟨*SF pdf builtin: procedures*⟩+≡
```
     subroutine pdf_builtin_complete_kinematics (sf_int, x, f, r, rb, map)
       class(pdf_builtin_t), intent(inout) :: sf_int
       real(default), dimension(:), intent(out) :: x
       real(default), intent(out) :: f
       real(default), dimension(:), intent(in) :: r
       real(default), dimension(:), intent(in) :: rb
       logical, intent(in) :: map
       real(default) :: xb1
       if (map) then
          call msg_fatal ("PDF builtin: map flag not supported")
       else
          x(1) = r(1)
          f = 1
       end if
       xb1 = 1 - x(1)
       call sf_int%split_momentum (x, xb1)
       select case (sf_int%status)
       case (SF_DONE_KINEMATICS)
          sf_int%x = x(1)
       case (SF_FAILED_KINEMATICS)
          sf_int%x = 0
          f = 0
       end select
     end subroutine pdf_builtin_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡
```
     procedure :: inverse_kinematics => pdf_builtin_inverse_kinematics
```

⟨*SF pdf builtin: procedures*⟩+≡
```
     subroutine pdf_builtin_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
       class(pdf_builtin_t), intent(inout) :: sf_int
       real(default), dimension(:), intent(in) :: x
```

1334

```
real(default), intent(out) :: f
real(default), dimension(:), intent(out) :: r
real(default), dimension(:), intent(out) :: rb
logical, intent(in) :: map
logical, intent(in), optional :: set_momenta
real(default) :: xb1
logical :: set_mom
set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
if (map) then
   call msg_fatal ("PDF builtin: map flag not supported")
else
   r(1) = x(1)
   f = 1
end if
xb1 = 1 - x(1)
rb = 1 - r
if (set_mom) then
   call sf_int%split_momentum (x, xb1)
   select case (sf_int%status)
   case (SF_DONE_KINEMATICS)
      sf_int%x = x(1)
   case (SF_FAILED_KINEMATICS)
      sf_int%x = 0
      f = 0
   end select
end if
end subroutine pdf_builtin_inverse_kinematics
```

10.12.5 Structure function

Once the scale is also known, we can actually call the PDF and set the values.
Contrary to LHAPDF, the wrapper already takes care of adjusting to the x and
Q bounds. Account for the Jacobian.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡
```
procedure :: apply => pdf_builtin_apply
```

⟨*SF pdf builtin: procedures*⟩+≡
```
subroutine pdf_builtin_apply (sf_int, scale)
  class(pdf_builtin_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default), dimension(-6:6) :: ff
  real(default) :: x, fph
  complex(default), dimension(:), allocatable :: fc
  associate (data => sf_int%data)
    sf_int%q = scale
    x = sf_int%x
    if (data%invert) then
       if (data%has_photon) then
          call pdf_evolve (data%id, x, scale, ff(6:-6:-1), fph)
       else
          call pdf_evolve (data%id, x, scale, ff(6:-6:-1))
       end if
    else
```

```
      if (data%has_photon) then
         call pdf_evolve (data%id, x, scale, ff, fph)
      else
         call pdf_evolve (data%id, x, scale, ff)
      end if
   end if
   if (data%has_photon) then
      allocate (fc (count ([data%mask, data%mask_photon])))
      fc = max (pack ([ff, fph], &
         [data%mask, data%mask_photon]), 0._default)
   else
      allocate (fc (count (data%mask)))
      fc = max (pack (ff, data%mask), 0._default)
   end if
 end associate
 call interaction_set_matrix_element (sf_int%interaction_t, fc)
 sf_int%status = SF_EVALUATED
end subroutine pdf_builtin_apply
```

10.12.6 Strong Coupling

Since the PDF codes provide a function for computing the running α_s value, we make this available as an implementation of the abstract `alpha_qcd_t` type, which is used for matrix element evaluation.

⟨*SF pdf builtin: public*⟩+≡
```
    public :: alpha_qcd_pdf_builtin_t
```

⟨*SF pdf builtin: types*⟩+≡
```
    type, extends (alpha_qcd_t) :: alpha_qcd_pdf_builtin_t
      type(string_t) :: pdfset_name
      integer :: pdfset_id = -1
    contains
    ⟨SF pdf builtin: alpha qcd: TBP⟩
    end type alpha_qcd_pdf_builtin_t
```

Output.

⟨*SF pdf builtin: alpha qcd: TBP*⟩≡
```
    procedure :: write => alpha_qcd_pdf_builtin_write
```

⟨*SF pdf builtin: procedures*⟩+≡
```
    subroutine alpha_qcd_pdf_builtin_write (object, unit)
      class(alpha_qcd_pdf_builtin_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A)")  "QCD parameters (pdf_builtin):"
      write (u, "(5x,A,A)")  "PDF set = ", char (object%pdfset_name)
      write (u, "(5x,A,I0)") "PDF ID  = ", object%pdfset_id
    end subroutine alpha_qcd_pdf_builtin_write
```

Calculation: the numeric ID selects the correct PDF set, which must be properly initialized.

⟨SF pdf builtin: alpha qcd: TBP⟩+≡
```
    procedure :: get => alpha_qcd_pdf_builtin_get
```

⟨SF pdf builtin: procedures⟩+≡
```
    function alpha_qcd_pdf_builtin_get (alpha_qcd, scale) result (alpha)
      class(alpha_qcd_pdf_builtin_t), intent(in) :: alpha_qcd
      real(default), intent(in) :: scale
      real(default) :: alpha
      alpha = pdf_alphas (alpha_qcd%pdfset_id, scale)
    end function alpha_qcd_pdf_builtin_get
```

Initialization. We need to access the (quasi-global) initialization status.

⟨SF pdf builtin: alpha qcd: TBP⟩+≡
```
    procedure :: init => alpha_qcd_pdf_builtin_init
```

⟨SF pdf builtin: procedures⟩+≡
```
    subroutine alpha_qcd_pdf_builtin_init (alpha_qcd, status, name, path)
      class(alpha_qcd_pdf_builtin_t), intent(out) :: alpha_qcd
      type(pdf_builtin_status_t), intent(inout) :: status
      type(string_t), intent(in) :: name
      type(string_t), intent(in) :: path
      alpha_qcd%pdfset_name = name
      alpha_qcd%pdfset_id = pdf_get_id (name)
      if (alpha_qcd%pdfset_id < 0) &
           call msg_fatal ("QCD parameter initialization: PDF set " &
           // char (name) // " is unknown")
      call pdf_init (status, alpha_qcd%pdfset_id, path)
    end subroutine alpha_qcd_pdf_builtin_init
```

10.12.7 Unit tests

⟨SF pdf builtin: public⟩+≡
```
    public :: sf_pdf_builtin_test
```

⟨SF pdf builtin: tests⟩≡
```
    subroutine sf_pdf_builtin_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF pdf builtin: execute tests⟩
    end subroutine sf_pdf_builtin_test
```

Test structure function data

Construct and display a test structure function data object.

⟨SF pdf builtin: execute tests⟩≡
```
    call test (sf_pdf_builtin_1, "sf_pdf_builtin_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF pdf builtin: tests*⟩+≡

```
    subroutine sf_pdf_builtin_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      type(pdg_array_t), dimension(1) :: pdg_out
      integer, dimension(:), allocatable :: pdg1
      class(sf_data_t), allocatable :: data
      type(pdf_builtin_status_t) :: status
      type(string_t) :: name

      write (u, "(A)")  "* Test output: sf_pdf_builtin_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
          &test structure function data"
      write (u, "(A)")

      write (u, "(A)")  "* Create empty data object"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QCD"), &
          var_str ("QCD.mdl"), os_data, model)
      pdg_in = PROTON

      allocate (pdf_builtin_data_t :: data)
      call data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize"
      write (u, "(A)")

      name = "CTEQ6L"

      select type (data)
      type is (pdf_builtin_data_t)
         call data%init (status, model, pdg_in, name, &
             os_data%pdf_builtin_datapath)
      end select

      call data%write (u)

      write (u, "(A)")

      write (u, "(1x,A)")  "Outgoing particle codes:"
      call data%get_pdg_out (pdg_out)
      pdg1 = pdg_out(1)
      write (u, "(2x,99(1x,I0))")  pdg1

      call model_list%final ()
      call syntax_model_file_final ()
```

1338

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_pdf_builtin_1"

  end subroutine sf_pdf_builtin_1
```

Test and probe structure function

Construct and display a structure function object based on the PDF builtin
structure function.

⟨*SF pdf builtin: execute tests*⟩+≡
```
    call test (sf_pdf_builtin_2, "sf_pdf_builtin_2", &
        "structure function instance", &
        u, results)
```

⟨*SF pdf builtin: tests*⟩+≡
```
  subroutine sf_pdf_builtin_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(pdf_builtin_status_t) :: status
    type(string_t) :: name
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_pdf_builtin_2"
    write (u, "(A)")  "*   Purpose: initialize and fill &
        &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list%read_model (var_str ("QCD"), &
        var_str ("QCD.mdl"), os_data, model)
    call flavor_init (flv, PROTON, model)
    pdg_in = PROTON

    call reset_interaction_counter ()

    name = "CTEQ6L"

    allocate (pdf_builtin_data_t :: data)
    select type (data)
```

1339

```
type is (pdf_builtin_data_t)
   call data%init (status, model, pdg_in, name, &
        os_data%pdf_builtin_datapath)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.5_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = interaction_get_momenta (sf_int%interaction_t, outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call interaction_set_momenta (sf_int%interaction_t, q, outgoing=.true.)
call sf_int%recover_x (x)
```

```
      write (u, "(A,9(1x,F10.7))")  "x =", x

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate for Q = 100 GeV"
      write (u, "(A)")

      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%apply (scale = 100._default)
      call sf_int%write (u)


      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_pdf_builtin_2"

    end subroutine sf_pdf_builtin_2
```

Strong Coupling

Test α_s as an implementation of the `alpha_qcd_t` abstract type.

⟨*SF pdf builtin: execute tests*⟩+≡
```
    call test (sf_pdf_builtin_3, "sf_pdf_builtin_3", &
         "running alpha_s", &
         u, results)
```

⟨*SF pdf builtin: tests*⟩+≡
```
    subroutine sf_pdf_builtin_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(pdf_builtin_status_t) :: status
      type(qcd_t) :: qcd
      type(string_t) :: name

      write (u, "(A)")  "* Test output: sf_pdf_builtin_3"
      write (u, "(A)")  "*   Purpose: initialize and evaluate alpha_s"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)

      name = "CTEQ6L"

      write (u, "(A)")  "* Initialize qcd object"
```

```
      write (u, "(A)")

      allocate (alpha_qcd_pdf_builtin_t :: qcd%alpha)
      select type (alpha => qcd%alpha)
      type is (alpha_qcd_pdf_builtin_t)
         call alpha%init (status, name, os_data%pdf_builtin_datapath)
      end select
      call qcd%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate for Q = 100"
      write (u, "(A)")

      write (u, "(1x,A,F8.5)")  "alpha = ", qcd%alpha%get (100._default)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call model_list%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_pdf_builtin_3"

    end subroutine sf_pdf_builtin_3
```

10.13   User Plugin for Structure Functions

This variant gives access to user-defined structure functions or spectra.

10.13.1   The module

⟨sf_user.f90⟩≡
  ⟨*File header*⟩

  module sf_user

    use iso_c_binding !NODEP!
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: FMT_17 !NODEP!
    use diagnostics !NODEP!
    use c_particles !NODEP!
    use lorentz !NODEP!
    use subevents
    use user_code_interface
    use pdg_arrays
    use models
    use flavors
    use helicities
    use colors
    use quantum_numbers

```
    use state_matrices
    use polarizations
    use interactions
    use sf_aux
    use sf_base
```

⟨Standard module head⟩

⟨SF user: public⟩

⟨SF user: types⟩

```
contains
```

⟨SF user: procedures⟩

```
end module sf_user
```

10.13.2   The user structure function data block

The data block holds the procedure pointers that are used for retrieving static
information, as well as the actual evaluation.

⟨SF user: public⟩≡
```
    public :: user_data_t
```

⟨SF user: types⟩≡
```
    type, extends(sf_data_t) :: user_data_t
      private
      type(string_t) :: name
      integer :: n_in
      integer :: n_out
      integer :: n_tot
      integer :: n_states
      integer :: n_col
      integer :: n_dim
      integer :: n_var
      integer, dimension(2) :: pdg_in
      type(model_t), pointer :: model => null ()
      procedure(user_int_info), nopass, pointer :: info => null ()
      procedure(user_int_mask), nopass, pointer :: mask => null ()
      procedure(user_int_state), nopass, pointer :: state => null ()
      procedure(user_int_kinematics), nopass, pointer :: kinematics => null ()
      procedure(user_int_evaluate), nopass, pointer :: evaluate => null ()
    contains
    ⟨SF user: user data: TBP⟩
    end type user_data_t
```

Assign procedure pointers from a dynamically loaded library, given the specified
`name`.

We have to distinguish three cases: (1) Both beams are affected, and the
user spectrum implements both beams. There is a single data object. (2) Both

beams are affected, and the user spectrum applies to single beams. Fill two different objects. (3) A single beam is affected.

⟨*SF User: public*⟩≡
```
    public :: sf_user_data_init
```

⟨*SF User: procedures*⟩≡
```
    subroutine sf_user_data_init (data, name, flv, model)
      type(sf_user_data_t), intent(out) :: data
      type(string_t), intent(in) :: name
      type(flavor_t), dimension(2), intent(in) :: flv
      type(model_t), intent(in), target :: model
      integer(c_int) :: n_in
      integer(c_int) :: n_out
      integer(c_int) :: n_states
      integer(c_int) :: n_col
      integer(c_int) :: n_dim
      integer(c_int) :: n_var
      data%name = name
      data%pdg_in = flavor_get_pdg (flv)
      data%model => model
      call c_f_procpointer (user_code_find_proc (name // "_info"), data%info)
      call c_f_procpointer (user_code_find_proc (name // "_mask"), data%mask)
      call c_f_procpointer (user_code_find_proc (name // "_state"), data%state)
      call c_f_procpointer &
           (user_code_find_proc (name // "_kinematics"), data%kinematics)
      call c_f_procpointer &
           (user_code_find_proc (name // "_evaluate"), data%evaluate)
      n_in = 1
      n_out = 2
      n_states = 1
      n_col = 2
      n_dim = 1
      n_var = 1
      call data%info (n_in, n_out, n_states, n_col, n_dim, n_var)
      data%n_in = n_in
      data%n_out = n_out
      data%n_tot = n_in + n_out
      data%n_states = n_states
      data%n_col = n_col
      data%n_dim = n_dim
      data%n_var = n_var
    end subroutine sf_user_data_init
```

Output

⟨*SF user: user data: TBP*⟩≡
```
    procedure :: write => user_data_write
```

⟨*SF user: procedures*⟩≡
```
    subroutine user_data_write (data, unit, verbose)
      class(user_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
```

1344

```
    write (u, "(1x,A,A)") "User structure function: ", char (data%name)
  end subroutine user_data_write
```

Retrieving contents

⟨*SF User: public*⟩+≡
```
  public :: sf_user_data_get_name
```

⟨*SF User: procedures*⟩+≡
```
  function sf_user_data_get_name (data) result (name)
    type(string_t) :: name
    type(sf_user_data_t), intent(in) :: data
    name = data%name
  end function sf_user_data_get_name
```

⟨*SF User: public*⟩+≡
```
  public :: sf_user_data_get_n_in
  public :: sf_user_data_get_n_out
  public :: sf_user_data_get_n_tot
  public :: sf_user_data_get_n_dim
  public :: sf_user_data_get_n_var
```

⟨*SF User: procedures*⟩+≡
```
  function sf_user_data_get_n_in (data) result (n_in)
    integer :: n_in
    type(sf_user_data_t), intent(in) :: data
    n_in = data%n_in
  end function sf_user_data_get_n_in

   function sf_user_data_get_n_out (data) result (n_out)
    integer :: n_out
    type(sf_user_data_t), intent(in) :: data
    n_out = data%n_out
  end function sf_user_data_get_n_out

  function sf_user_data_get_n_tot (data) result (n_tot)
    integer :: n_tot
    type(sf_user_data_t), intent(in) :: data
    n_tot = data%n_tot
  end function sf_user_data_get_n_tot

  function sf_user_data_get_n_dim (data) result (n_dim)
    integer :: n_dim
    type(sf_user_data_t), intent(in) :: data
    n_dim = data%n_dim
  end function sf_user_data_get_n_dim

  function sf_user_data_get_n_var (data) result (n_var)
    integer :: n_var
    type(sf_user_data_t), intent(in) :: data
    n_var = data%n_var
  end function sf_user_data_get_n_var
```

10.13.3 The interaction

We fill the interaction by looking up the table of states using the interface
functions.

For particles which have a known flavor (as indicated by the mask), we
compute the mass squared, so we can use it for the invariant mass of the particle
objects.

⟨*SF user: user: TBP*⟩≡
```
    procedure :: init => user_init
```

⟨*SF user: procedures*⟩+≡
```
    subroutine user_init (sf_int, data)
      !!! JRR: WK please check (#529)
      class(user_t), intent(out) :: sf_int
      class(sf_data_t), intent(in), target :: data
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
      integer, dimension(:), allocatable :: hel_lock
      integer(c_int) :: m_flv, m_hel, m_col, i_lock
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      integer(c_int) :: f, h
      integer(c_int), dimension(:), allocatable :: c
      type(flavor_t) :: flv
      type(helicity_t) :: hel
      type(color_t) :: col
      integer :: i, s
      integer(c_int) :: i_prt, i_state
      select type (data)
      type is (user_data_t)
         allocate (mask (data%n_tot))
         allocate (hel_lock (data%n_tot))
         allocate (qn (data%n_tot))
         allocate (c (data%n_col))
         do i = 1, size (mask)
            i_prt = i
            m_flv = 0;  m_col = 0;  m_hel = 0;  i_lock = 0
            call data%mask (i_prt, m_flv, m_col, m_hel, i_lock)
            mask(i) = &
                 new_quantum_numbers_mask (m_flv /= 0, m_col /= 0, m_hel /= 0)
            hel_lock(i) = i_lock
         end do
         !!! JRR: WK please check (#529)
         !!! Will have to be filled in later.
         ! call sf_int%base_init (mask, &
         !      hel_lock = hel_lock)
         call interaction_init &
              (sf_int%interaction_t, data%n_in, 0, data%n_out, mask=mask, &
              hel_lock=hel_lock, set_relations=.true.)
         do s = 1, data%n_states
            i_state = s
            do i = 1, data%n_tot
               i_prt = i
               f = 0;  h = 0;  c = 0
               call data%state (i_state, i_prt, f, h, c)
               if (m_flv == 0) then
```

```
                 call flavor_init (flv, int (f), data%model)
              else
                 call flavor_init (flv)
              end if
              if (m_hel == 0) then
                 call helicity_init (hel, int (h))
              else
                 call helicity_init (hel)
              end if
              if (m_col == 0) then
                 call color_init_from_array (col, int (c))
              else
                 call color_init (col)
              end if
              call quantum_numbers_init (qn(i), flv, col, hel)
           end do
           call interaction_add_state (sf_int%interaction_t, qn)
        end do
        call interaction_freeze (sf_int%interaction_t)
        !!! JRR: WK please check (#529)
        !!! What has to be inserted here?
        ! call sf_int%set_incoming (??)
        ! call sf_int%set_radiated (??)
        ! call sf_int%set_outgoing (??)
        sf_int%status = SF_INITIAL
     end select
   end subroutine user_init
```

Allocate the interaction record.

⟨*SF user: user data: TBP*⟩+≡
```
   procedure :: allocate_sf_int => user_data_allocate_sf_int
```

⟨*SF user: procedures*⟩+≡
```
   subroutine user_data_allocate_sf_int (data, sf_int)
     class(user_data_t), intent(in) :: data
     class(sf_int_t), intent(inout), allocatable :: sf_int
     allocate (user_t :: sf_int)
   end subroutine user_data_allocate_sf_int
```

The number of parameters is one. We do not generate transverse momentum.

⟨*SF user: user data: TBP*⟩+≡
```
   procedure :: get_n_par => user_data_get_n_par
```

⟨*SF user: procedures*⟩+≡
```
   function user_data_get_n_par (data) result (n)
     class(user_data_t), intent(in) :: data
     integer :: n
     n = data%n_var
   end function user_data_get_n_par
```

Return the outgoing particle PDG codes. This has to be inferred from the states (right?). JRR: WK please check.

⟨*SF user: user data: TBP*⟩+≡
```
   procedure :: get_pdg_out => user_data_get_pdg_out
```

1347

⟨*SF user: procedures*⟩+≡
```
    subroutine user_data_get_pdg_out (data, pdg_out)
      class(user_data_t), intent(in) :: data
      type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
      !!! JRR: WK please check (#529)
      !!! integer :: n, np, i
      !!! n = count (data%mask)
      !!! np = 0;  if (data%has_photon .and. data%mask_photon)  np = 1
      !!! allocate (pdg_out (n + np))
      !!! pdg_out(1:n) = pack ([(i, i = -6, 6)], data%mask)
      !!! if (np == 1)  pdg_out(n+np) = PHOTON
    end subroutine user_data_get_pdg_out
```

10.13.4   The user structure function

For maximal flexibility, user structure functions separate kinematics from dynamics just as the PDF interface does. (JRR: Ok, I guess this now done for all structure functions, right?) We create `c_prt_t` particle objects from the incoming momenta (all other quantum numbers are irrelevant) and call the user-supplied kinematics function to compute the outgoing momenta, along with other variables that will be needed for matrix element evaluation. If known, we use the mass squared computed above. !!! JRR: WK please check (#529) I don't know actually whether this really fits into the setup done by WK.

⟨*SF user: types*⟩+≡
```
    !!! JRR: WK please check (#529)
    type, extends (sf_int_t) :: user_t
      type(user_data_t), pointer :: data => null ()
      real(default) :: x = 0
      real(default) :: q = 0
     contains
     ⟨SF user: user: TBP⟩
    end type user_t
```

Type string: display the name of the user structure function.

⟨*SF user: user: TBP*⟩+≡
```
    procedure :: type_string => user_type_string
```

⟨*SF user: procedures*⟩+≡
```
    function user_type_string (object) result (string)
      class(user_t), intent(in) :: object
      type(string_t) :: string
      if (associated (object%data)) then
         string = "User structure function: " // object%data%name
      else
         string = "User structure function: [undefined]"
      end if
    end function user_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF user: user: TBP*⟩+≡
```
    procedure :: write => user_write
```

1348

⟨*SF user: procedures*⟩+≡

```
subroutine user_write (object, unit, testflag)
  !!! JRR: WK please check (#529)
  !!! Guess these variables do not exist for user strfun (?)
  class(user_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
           write (u, "(3x,A," // FMT_17 // ")")  "Q =", object%q
        end if
     end if
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "User structure function data: [undefined]"
  end if
end subroutine user_write
```

10.13.5 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF user: user: TBP*⟩+≡

```
procedure :: complete_kinematics => user_complete_kinematics
```

⟨*SF user: procedures*⟩+≡

```
subroutine user_complete_kinematics (sf_int, x, f, r, rb, map)
  !!! JRR: WK please check (#529)
  !!! This cannot be correct, as the CIRCE1 structure function has
  !!! twice the variables (2->4 instead of 1->2 splitting)
  class(user_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default) :: xb1
  if (map) then
     call msg_fatal ("User structure function: map flag not supported")
  else
     x(1) = r(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  call sf_int%split_momentum (x, xb1)
```

```
      select case (sf_int%status)
      case (SF_DONE_KINEMATICS)
         sf_int%x = x(1)
      case (SF_FAILED_KINEMATICS)
         sf_int%x = 0
         f = 0
      end select
    end subroutine user_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF user: user: TBP*⟩+≡
```
    procedure :: inverse_kinematics => user_inverse_kinematics
```

⟨*SF user: procedures*⟩+≡
```
    subroutine user_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
      !!! JRR: WK please check (#529)
      !!! This cannot be correct, as the CIRCE1 structure function has
      !!! twice the variables (2->4 instead of 1->2 splitting)
      class(user_t), intent(inout) :: sf_int
      real(default), dimension(:), intent(in) :: x
      real(default), intent(out) :: f
      real(default), dimension(:), intent(out) :: r
      real(default), dimension(:), intent(out) :: rb
      logical, intent(in) :: map
      logical, intent(in), optional :: set_momenta
      real(default) :: xb1
      logical :: set_mom
      set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
      if (map) then
         call msg_fatal ("User structure function: map flag not supported")
      else
         r(1) = x(1)
         f = 1
      end if
      xb1 = 1 - x(1)
      rb = 1 - r
      if (set_mom) then
         call sf_int%split_momentum (x, xb1)
         select case (sf_int%status)
         case (SF_DONE_KINEMATICS)
            sf_int%x = x(1)
         case (SF_FAILED_KINEMATICS)
            sf_int%x = 0
            f = 0
         end select
      end if
    end subroutine user_inverse_kinematics
```

⟨*SF User: public*⟩+≡
```
    public :: interaction_set_kinematics_sf_user
```

1350

⟨*SF User: procedures*⟩+≡
```
subroutine interaction_set_kinematics_sf_user (int, x, r, data)
  type(interaction_t), intent(inout) :: int
  real(default), dimension(:), intent(out) :: x
  real(default), dimension(:), intent(in) :: r
  type(sf_user_data_t), intent(in) :: data
  type(vector4_t), dimension(data%n_in) :: p_in
  type(vector4_t), dimension(data%n_out) :: p_out
  type(c_prt_t), dimension(data%n_in) :: prt_in
  type(c_prt_t), dimension(data%n_out) :: prt_out
  real(c_double), dimension(data%n_var) :: xval
  call interaction_get_momenta_sub (int, p_in, outgoing=.false.)
  prt_in = vector4_to_c_prt (p_in)
  prt_in%type = PRT_INCOMING
  call data%kinematics (prt_in, real (r, c_double), prt_out, xval)
  x = xval
  p_out = vector4_from_c_prt (prt_out)
  call interaction_set_momenta (int, p_out, outgoing=.true.)
end subroutine interaction_set_kinematics_sf_user
```

The matrix-element evaluation may require a scale parameter, therefore this routine is separate. We take the variables computed above together with the event energy scale and call the user function that computes the matrix elements.

⟨*SF user: user: TBP*⟩+≡
```
procedure :: apply => user_apply
```

⟨*SF user: procedures*⟩+≡
```
subroutine user_apply (sf_int, scale) !, x, data)
  !!! JRR: WK please check (#529)
  class(user_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default), dimension(:), allocatable :: x
  real(c_double), dimension(sf_int%data%n_states) :: fval
  complex(default), dimension(sf_int%data%n_states) :: fc
  associate (data => sf_int%data)
    !!! This is wrong, has to be replaced
    ! allocate (x, size (sf_int%x)))
    x = sf_int%x
    call data%evaluate (real (x, c_double), real (scale, c_double), fval)
    fc = fval
    call interaction_set_matrix_element (sf_int%interaction_t, fc)
  end associate
  sf_int%status = SF_EVALUATED
end subroutine user_apply
```

10.14   HOPPET interface

Interface to the HOPPET wrapper necessary to perform the LO vs. NLO matching of processes containing an initial b quark.

⟨`hoppet_interface.f90`⟩≡
  ⟨*File header*⟩

```
module hoppet_interface
```

⟨*Standard module head*⟩

```
  public :: hoppet_init, hoppet_eval

contains

  subroutine hoppet_init ()
    external InitForWhizard
    call InitForWhizard ()
  end subroutine hoppet_init

  subroutine hoppet_eval (x, q, f)
    double precision, intent(in)  :: x, q
    double precision, intent(out) :: f(-6:6)
    external EvalForWhizard
    call EvalForWhizard (x, q, f)
  end subroutine hoppet_eval

end module hoppet_interface
```

Chapter 11

Phase space and hard matrix elements

These modules contain the internal representation and evaluation of phase space and the interface to (hard-)process evaluation.

mappings Generate invariant masses and decay angles from given random numbers (or the inverse operation). Each mapping pertains to a particular node in a phase-space tree. Different mappings account for uniform distributions, resonances, zero-mass behavior, and so on.

phs_trees Phase space parameterizations for scattering processes are defined recursively as if there was an initial particle decaying. This module sets up a representation in terms of abstract trees, where each node gets a unique binary number. Each tree is stored as an array of branches, where integers indicate the connections. This emulates pointers in a transparent way. Real pointers would also be possible, but seem to be less efficient for this particular case.

phs_forests The type defined by this module collects the decay trees corresponding to a given process and the applicable mappings. To set this up, a file is read which is either written by the user or by the **cascades** module functions. The module also contains the routines that evaluate phase space, i.e., generate momenta from random numbers and back.

cascades This module is a Feynman diagram generator with the particular purpose of finding the phase space parameterizations best suited for a given process. It uses a model file to set up the possible vertices, generates all possible diagrams, identifies resonances and singularities, and simplifies the list by merging equivalent diagrams and dropping irrelevant ones. This process can be controlled at several points by user-defined parameters. Note that it depends on the particular values of particle masses, so it cannot be done before reading the input file.

11.1 Process data block

We define a simple transparent type that contains universal constant process data. We will reference objects of this type for the phase-space setup, for interfacing with process libraries, for implementing matrix-element generation, and in the master process-handling module.

⟨process_constants.f90⟩≡

  ⟨*File header*⟩

  `module process_constants`

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩

  ⟨*Standard module head*⟩

  ⟨*Process constants: public*⟩

  ⟨*Process constants: types*⟩

  `end module process_constants`

The data type is just a block of public objects, only elementary types, no type-bound procedures.

⟨*Process constants: public*⟩≡

```
    public :: process_constants_t
```

⟨*Process constants: types*⟩≡

```
    type :: process_constants_t
       type(string_t) :: id
       type(string_t) :: model_name
       character(32) :: md5sum = ""
       logical :: openmp_supported = .false.
       integer :: n_in  = 0
       integer :: n_out = 0
       integer :: n_flv = 0
       integer :: n_hel = 0
       integer :: n_col = 0
       integer :: n_cin = 0
       integer :: n_cf  = 0
       integer, dimension(:,:), allocatable :: flv_state
       integer, dimension(:,:), allocatable :: hel_state
       integer, dimension(:,:,:), allocatable :: col_state
       logical, dimension(:,:), allocatable :: ghost_flag
       complex(default), dimension(:), allocatable :: color_factors
       integer, dimension(:,:), allocatable :: cf_index
    end type process_constants_t
```

11.2 Abstract phase-space module

In this module we define an abstract base type (and a trivial test implementation) for multi-channel phase-space parameterizations.

⟨phs_base.f90⟩≡
  ⟨*File header*⟩

  module phs_base

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: FMT_19 !NODEP!
    use constants !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use os_interface
    use md5
    use variables
    use models
    use flavors
    use process_constants
    use sf_mappings
    use sf_base

  ⟨*Standard module head*⟩

  ⟨*PHS base: public*⟩

  ⟨*PHS base: types*⟩

  ⟨*PHS base: interfaces*⟩

  ⟨*PHS base: test types*⟩

  contains

  ⟨*PHS base: procedures*⟩

  ⟨*PHS base: tests*⟩

  end module phs_base

11.2.1 Phase-space channels

The kinematics configuration may generate multiple parameterizations of phase space. Some of those have specific properties, such as a resonance in the s channel.

Channel properties

This is the abstract type for the channel properties. We need them as a data transfer container, so everything is public and transparent.

⟨*PHS base: public*⟩≡
```
public :: channel_prop_t
```

⟨*PHS base: types*⟩≡
```
type, abstract :: channel_prop_t
 contains
   procedure (channel_prop_to_string), deferred :: to_string
   generic :: operator (==) => is_equal
   procedure (channel_eq), deferred :: is_equal
end type channel_prop_t
```

⟨*PHS base: interfaces*⟩≡
```
abstract interface
   function channel_prop_to_string (object) result (string)
     import
     class(channel_prop_t), intent(in) :: object
     type(string_t) :: string
   end function channel_prop_to_string
end interface
```

⟨*PHS base: interfaces*⟩+≡
```
abstract interface
   function channel_eq (prop1, prop2) result (flag)
     import
     class(channel_prop_t), intent(in) :: prop1, prop2
     logical :: flag
   end function channel_eq
end interface
```

Here is a resonance as a channel property. Mass and width are stored here in physical units.

⟨*PHS base: public*⟩+≡
```
public :: resonance_t
```

⟨*PHS base: types*⟩+≡
```
type, extends (channel_prop_t) :: resonance_t
   real(default) :: mass = 0
   real(default) :: width = 0
 contains
   procedure :: to_string => resonance_to_string
   procedure :: is_equal => resonance_is_equal
end type resonance_t
```

Print mass and width.

⟨*PHS base: procedures*⟩≡
```
function resonance_to_string (object) result (string)
   class(resonance_t), intent(in) :: object
   type(string_t) :: string
   character(32) :: buffer
```

```
       string = "resonant: m ="
       write (buffer, "(" // FMT_19 // ")")  object%mass
       string = string // trim (buffer) // " GeV, w ="
       write (buffer, "(" // FMT_19 // ")")  object%width
       string = string // trim (buffer) // " GeV"
    end function resonance_to_string
```

Equality.

⟨*PHS base: procedures*⟩+≡
```
    function resonance_is_equal (prop1, prop2) result (flag)
      class(resonance_t), intent(in) :: prop1
      class(channel_prop_t), intent(in) :: prop2
      logical :: flag
      select type (prop2)
      type is (resonance_t)
         flag = prop1%mass == prop2%mass .and. prop1%width == prop2%width
      class default
         flag = .false.
      end select
    end function resonance_is_equal
```

This is the limiting case of a resonance, namely an on-shell particle. We just store the mass in physical units.

⟨*PHS base: public*⟩+≡
```
    public :: on_shell_t
```

⟨*PHS base: types*⟩+≡
```
    type, extends (channel_prop_t) :: on_shell_t
      real(default) :: mass = 0
     contains
       procedure :: to_string => on_shell_to_string
       procedure :: is_equal => on_shell_is_equal
    end type on_shell_t
```

Print mass and width.

⟨*PHS base: procedures*⟩+≡
```
    function on_shell_to_string (object) result (string)
      class(on_shell_t), intent(in) :: object
      type(string_t) :: string
      character(32) :: buffer
      string = "on shell: m ="
      write (buffer, "(" // FMT_19 // ")")  object%mass
      string = string // trim (buffer) // " GeV"
    end function on_shell_to_string
```

Equality.

⟨*PHS base: procedures*⟩+≡
```
    function on_shell_is_equal (prop1, prop2) result (flag)
      class(on_shell_t), intent(in) :: prop1
      class(channel_prop_t), intent(in) :: prop2
      logical :: flag
      select type (prop2)
```

```
      type is (on_shell_t)
         flag = prop1%mass == prop2%mass
      class default
         flag = .false.
      end select
   end function on_shell_is_equal
```

Channel equivalences

This type describes an equivalence. The current channel is equivalent to channel
`c`. The equivalence involves a permutation `perm` of integration dimensions and,
within each integration dimension, a mapping `mode`.

⟨*PHS base: types*⟩+≡
```
   type :: phs_equivalence_t
      integer :: c = 0
      integer, dimension(:), allocatable :: perm
      integer, dimension(:), allocatable :: mode
    contains
    ⟨PHS base: phs equivalence: TBP⟩
   end type phs_equivalence_t
```

The mapping modes are

⟨*PHS base: types*⟩+≡
```
   integer, parameter, public :: &
        EQ_IDENTITY = 0, EQ_INVERT = 1, EQ_SYMMETRIC = 2, EQ_INVARIANT = 3
```

In particular, if a channel is equivalent to itself in the `EQ_SYMMETRIC` mode, the
integrand can be assumed to be symmetric w.r.t. a reflection $x \to 1 - x$ of the
correponding integration variable.

These are the associated tags, for output:

⟨*PHS base: types*⟩+≡
```
   character, dimension(0:3), parameter :: TAG = ["+", "-", ":", "x"]
```

Write an equivalence.

⟨*PHS base: phs equivalence: TBP*⟩≡
```
   procedure :: write => phs_equivalence_write
```

⟨*PHS base: procedures*⟩+≡
```
   subroutine phs_equivalence_write (object, unit)
     class(phs_equivalence_t), intent(in) :: object
     integer, intent(in), optional :: unit
     integer :: u, j
     u = output_unit (unit)
     write (u, "(5x,'=',1x,I0,1x)", advance = "no")  object%c
     if (allocated (object%perm)) then
        write (u, "(A)", advance = "no")  "("
        do j = 1, size (object%perm)
           if (j > 1)  write (u, "(1x)", advance = "no")
           write (u, "(I0,A1)", advance = "no") &
                object%perm(j), TAG(object%mode(j))
        end do
```

```
      write (u, "(A)")  ")"
   else
      write (u, "(A)")
   end if
end subroutine phs_equivalence_write
```

Initialize an equivalence. This allocates the `perm` and `mode` arrays with equal size.

⟨*PHS base: phs equivalence: TBP*⟩+≡
```
procedure :: init => phs_equivalence_init
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_equivalence_init (eq, n_dim)
  class(phs_equivalence_t), intent(out) :: eq
  integer, intent(in) :: n_dim
  allocate (eq%perm (n_dim), source = 0)
  allocate (eq%mode (n_dim), source = EQ_IDENTITY)
end subroutine phs_equivalence_init
```


Channel objects

The channel entry holds (optionally) specific properties.

    `sf_channel` is the structure-function channel that corresponds to this phase-space channel. The structure-function channel may be set up with a specific mapping that depends on the phase-space channel properties. (The default setting is to leave the properties empty.)

⟨*PHS base: public*⟩+≡
```
public :: phs_channel_t
```

⟨*PHS base: types*⟩+≡
```
type :: phs_channel_t
  class(channel_prop_t), allocatable :: prop
  integer :: sf_channel = 1
  type(phs_equivalence_t), dimension(:), allocatable :: eq
 contains
 ⟨PHS base: phs channel: TBP⟩
end type phs_channel_t
```

Output.

⟨*PHS base: phs channel: TBP*⟩≡
```
procedure :: write => phs_channel_write
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_write (object, unit)
  class(phs_channel_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, j
  u = output_unit (unit)
  write (u, "(1x,I0)", advance="no") object%sf_channel
  if (allocated (object%prop)) then
     write (u, "(1x,A)")  char (object%prop%to_string ())
  else
```

```
      write (u, *)
   end if
   if (allocated (object%eq)) then
      do j = 1, size (object%eq)
         call object%eq(j)%write (u)
      end do
   end if
end subroutine phs_channel_write
```

Identify the channel with an s-channel resonance.

⟨PHS base: phs channel: TBP⟩+≡
```
procedure :: set_resonant => channel_set_resonant
```

⟨PHS base: procedures⟩+≡
```
subroutine channel_set_resonant (channel, mass, width)
  class(phs_channel_t), intent(inout) :: channel
  real(default), intent(in) :: mass, width
  allocate (resonance_t :: channel%prop)
  select type (prop => channel%prop)
  type is (resonance_t)
     prop%mass = mass
     prop%width = width
  end select
end subroutine channel_set_resonant
```

Identify the channel with an on-shell particle.

⟨PHS base: phs channel: TBP⟩+≡
```
procedure :: set_on_shell => channel_set_on_shell
```

⟨PHS base: procedures⟩+≡
```
subroutine channel_set_on_shell (channel, mass)
  class(phs_channel_t), intent(inout) :: channel
  real(default), intent(in) :: mass
  allocate (on_shell_t :: channel%prop)
  select type (prop => channel%prop)
  type is (on_shell_t)
     prop%mass = mass
  end select
end subroutine channel_set_on_shell
```

11.2.2 Property collection

We can set up a list of all distinct channel properties for a given set of channels.

⟨PHS base: public⟩+≡
```
public :: phs_channel_collection_t
```

⟨PHS base: types⟩+≡
```
type :: prop_entry_t
   integer :: i = 0
   class(channel_prop_t), allocatable :: prop
   type(prop_entry_t), pointer :: next => null ()
end type prop_entry_t
```

```
type :: phs_channel_collection_t
   integer :: n = 0
   type(prop_entry_t), pointer :: first => null ()
 contains
 ⟨PHS base: phs channel collection: TBP⟩
end type phs_channel_collection_t
```

Finalizer for the list.

⟨*PHS base: phs channel collection: TBP*⟩≡
```
procedure :: final => phs_channel_collection_final
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_collection_final (object)
  class(phs_channel_collection_t), intent(inout) :: object
  type(prop_entry_t), pointer :: entry
  do while (associated (object%first))
     entry => object%first
     object%first => entry%next
     deallocate (entry)
  end do
end subroutine phs_channel_collection_final
```

Output.

Note: eliminating the `string` auxiliary triggers an ICE in gfortran 4.7.2.

⟨*PHS base: phs channel collection: TBP*⟩+≡
```
procedure :: write => phs_channel_collection_write
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_collection_write (object, unit)
  class(phs_channel_collection_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(prop_entry_t), pointer :: entry
  type(string_t) :: string
  integer :: u
  u = output_unit (unit)
  entry => object%first
  do while (associated (entry))
     if (allocated (entry%prop)) then
        string = entry%prop%to_string ()
        write (u, "(1x,I0,1x,A)")  entry%i, char (string)
     else
        write (u, "(1x,I0)")  entry%i
     end if
     entry => entry%next
  end do
end subroutine phs_channel_collection_write
```

Push a new property to the stack if it is not yet included. Simultaneously, set the `sf_channel` entry in the phase-space channel object to the index of the matching entry, or the new entry if there was no match.

⟨*PHS base: phs channel collection: TBP*⟩+≡
```
procedure :: push => phs_channel_collection_push
```

⟨PHS base: procedures⟩+≡

```
subroutine phs_channel_collection_push (coll, channel)
  class(phs_channel_collection_t), intent(inout) :: coll
  type(phs_channel_t), intent(inout) :: channel
  type(prop_entry_t), pointer :: entry, new
  if (associated (coll%first)) then
     entry => coll%first
     do
        if (allocated (entry%prop)) then
           if (allocated (channel%prop)) then
              if (entry%prop == channel%prop) then
                 channel%sf_channel = entry%i
                 return
              end if
           end if
        else if (.not. allocated (channel%prop)) then
           channel%sf_channel = entry%i
           return
        end if
        if (associated (entry%next)) then
           entry => entry%next
        else
           exit
        end if
     end do
     allocate (new)
     entry%next => new
  else
     allocate (new)
     coll%first => new
  end if
  coll%n = coll%n + 1
  new%i = coll%n
  channel%sf_channel = new%i
  if (allocated (channel%prop)) then
     allocate (new%prop, source = channel%prop)
  end if
end subroutine phs_channel_collection_push
```

Return the number of collected distinct channels.

⟨PHS base: phs channel collection: TBP⟩+≡

```
procedure :: get_n => phs_channel_collection_get_n
```

⟨PHS base: procedures⟩+≡

```
function phs_channel_collection_get_n (coll) result (n)
  class(phs_channel_collection_t), intent(in) :: coll
  integer :: n
  n = coll%n
end function phs_channel_collection_get_n
```

Return a specific channel (property object).

⟨PHS base: phs channel collection: TBP⟩+≡

```
procedure :: get_entry => phs_channel_collection_get_entry
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_channel_collection_get_entry (coll, i, prop)
      class(phs_channel_collection_t), intent(in) :: coll
      integer, intent(in) :: i
      class(channel_prop_t), intent(out), allocatable :: prop
      type(prop_entry_t), pointer :: entry
      integer :: k
      if (i > 0 .and. i <= coll%n) then
         entry => coll%first
         do k = 2, i
            entry => entry%next
         end do
         if (allocated (entry%prop)) then
            if (allocated (prop))  deallocate (prop)
            allocate (prop, source = entry%prop)
         end if
      else
         call msg_bug ("PHS channel collection: get entry: illegal index")
      end if
    end subroutine phs_channel_collection_get_entry
```

11.2.3   Kinematics configuration

Here, we store the universal information that is specifically relevant for phase-space generation. It is a subset of the process data, supplemented by basic information on phase-space parameterization channels.

A concrete implementation will contain more data, that describe the phase space in detail.

MD5 sums: the phase space setup depends on the process, it depends on the model parameters (the masses, that is), and on the configuration parameters. (It doesn't depend on the QCD setup.)

⟨*PHS base: public*⟩+≡
```
    public :: phs_config_t
```

⟨*PHS base: types*⟩+≡
```
    type, abstract :: phs_config_t
      ! private
      type(string_t) :: id
      integer :: n_in = 0
      integer :: n_out = 0
      integer :: n_tot = 0
      integer :: n_state = 0
      integer :: n_par = 0
      integer :: n_channel = 0
      real(default) :: sqrts = 0
      logical :: sqrts_fixed = .true.
      logical :: cm_frame = .true.
      logical :: azimuthal_dependence = .false.
      integer, dimension(:), allocatable :: dim_flat
      logical :: provides_equivalences = .false.
      logical :: provides_chains = .false.
      logical :: vis_channels = .false.
```

```
      integer, dimension(:), allocatable :: chain
      type(model_t), pointer :: model => null ()
      type(flavor_t), dimension(:,:), allocatable :: flv
      type(phs_channel_t), dimension(:), allocatable :: channel
      character(32) :: md5sum_process = ""
      character(32) :: md5sum_model_par = ""
      character(32) :: md5sum_phs_config = ""
    contains
    ⟨PHS base: phs config: TBP⟩
    end type phs_config_t
```

Finalizer, deferred.

⟨*PHS base: phs config: TBP*⟩≡
```
    procedure (phs_config_final), deferred :: final
```

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
      subroutine phs_config_final (object)
        import
        class(phs_config_t), intent(inout) :: object
      end subroutine phs_config_final
    end interface
```

Output. We provide an implementation for the output of the base-type contents and an interface for the actual write method.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure (phs_config_write), deferred :: write
    procedure :: base_write => phs_config_write
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_config_write (object, unit)
      class(phs_config_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i, j
      u = output_unit (unit)
      write (u, "(3x,A,A,A)") "ID       = '", char (object%id), "'"
      write (u, "(3x,A,I0)")  "n_in     = ", object%n_in
      write (u, "(3x,A,I0)")  "n_out    = ", object%n_out
      write (u, "(3x,A,I0)")  "n_tot    = ", object%n_tot
      write (u, "(3x,A,I0)")  "n_state  = ", object%n_state
      write (u, "(3x,A,I0)")  "n_par    = ", object%n_par
      write (u, "(3x,A,I0)")  "n_channel = ", object%n_channel
      write (u, "(3x,A," // FMT_19 // ")")  "sqrts    = ", object%sqrts
      write (u, "(3x,A,L1)")  "s_fixed  = ", object%sqrts_fixed
      write (u, "(3x,A,L1)")  "cm_frame = ", object%cm_frame
      write (u, "(3x,A,L1)")  "azim.dep. = ", object%azimuthal_dependence
      if (allocated (object%dim_flat)) then
         write (u, "(3x,A,I0)")  "flat dim. = ", object%dim_flat
      end if
      write (u, "(1x,A)")  "Flavor combinations:"
      do i = 1, object%n_state
         write (u, "(3x,I0,':')", advance="no")  i
         do j = 1, object%n_tot
            write (u, "(1x,A)", advance="no") &
```

```
              char (flavor_get_name (object%flv(j,i)))
       end do
       write (u, *)
    end do
    if (allocated (object%channel)) then
       write (u, "(1x,A)")  "Phase-space / structure-function channels:"
       do i = 1, object%n_channel
          write (u, "(3x,I0,':')", advance="no") i
          call object%channel(i)%write (u)
       end do
    end if
    if (object%md5sum_process /= "") then
       write (u, "(3x,A,A,A)") "MD5 sum (process)   = '", &
            object%md5sum_process, "'"
    end if
    if (object%md5sum_model_par /= "") then
       write (u, "(3x,A,A,A)") "MD5 sum (model par)  = '", &
            object%md5sum_model_par, "'"
    end if
    if (object%md5sum_phs_config /= "") then
       write (u, "(3x,A,A,A)") "MD5 sum (phs config) = '", &
            object%md5sum_phs_config, "'"
    end if
  end subroutine phs_config_write
```

Similarly, a basic initializer and an interface. The model pointer is taken as an argument; we may verify that this has the expected model name.

The intent is `inout`. We want to be able to set parameters in advance.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: init => phs_config_init
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_config_init (phs_config, data, model)
      class(phs_config_t), intent(inout) :: phs_config
      type(process_constants_t), intent(in) :: data
      type(model_t), intent(in), target :: model
      integer :: i, j
      phs_config%id = data%id
      phs_config%n_in  = data%n_in
      phs_config%n_out = data%n_out
      phs_config%n_tot = data%n_in + data%n_out
      phs_config%n_state = data%n_flv
      if (data%model_name == model%get_name ()) then
         phs_config%model => model
      else
         call msg_bug ("phs_config_init: model name mismatch")
      end if
      allocate (phs_config%flv (phs_config%n_tot, phs_config%n_state))
      do i = 1, phs_config%n_state
         do j = 1, phs_config%n_tot
            call flavor_init (phs_config%flv(j,i), data%flv_state(j,i), &
                 phs_config%model)
         end do
      end do
```

1365

```
        phs_config%md5sum_process = data%md5sum
    end subroutine phs_config_init
```

This procedure should complete the phase-space configuration. We need the
sqrts value as overall scale, which is known only after the beams have been de-
fined. The procedure should determine the number of channels, their properties
(if any), and allocate and fill the channel array accordingly.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure (phs_config_configure), deferred :: configure
```

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
       subroutine phs_config_configure (phs_config, sqrts, &
            sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch)
         import
         class(phs_config_t), intent(inout) :: phs_config
         real(default), intent(in) :: sqrts
         logical, intent(in), optional :: sqrts_fixed
         logical, intent(in), optional :: cm_frame
         logical, intent(in), optional :: azimuthal_dependence
         logical, intent(in), optional :: rebuild
         logical, intent(in), optional :: ignore_mismatch
       end subroutine phs_config_configure
    end interface
```

Manually assign structure-function channel indices to the phase-space channel
objects. (Used by a test routine.)

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: set_sf_channel => phs_config_set_sf_channel
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_config_set_sf_channel (phs_config, sf_channel)
      class(phs_config_t), intent(inout) :: phs_config
      integer, dimension(:), intent(in) :: sf_channel
      phs_config%channel%sf_channel = sf_channel
    end subroutine phs_config_set_sf_channel
```

Collect new channels not yet in the collection from this phase-space configura-
tion object. At the same time, assign structure-function channels.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: collect_channels => phs_config_collect_channels
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_config_collect_channels (phs_config, coll)
      class(phs_config_t), intent(inout) :: phs_config
      type(phs_channel_collection_t), intent(inout) :: coll
      integer :: c
      do c = 1, phs_config%n_channel
         call coll%push (phs_config%channel(c))
      end do
    end subroutine phs_config_collect_channels
```

Compute the MD5 sum. We abuse the `write` method. In type implementations, `write` should only display information that is relevant for the MD5 sum. The data include the process MD5 sum which is taken from the process constants, and the MD5 sum of the model parameters. This may change, so it is computed here.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: compute_md5sum => phs_config_compute_md5sum
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_config_compute_md5sum (phs_config)
      class(phs_config_t), intent(inout) :: phs_config
      integer :: u
      phs_config%md5sum_model_par = model_get_parameters_md5sum (phs_config%model)
      phs_config%md5sum_phs_config = ""
      u = free_unit ()
      open (u, status = "scratch", action = "readwrite")
      call phs_config%write (u)
      rewind (u)
      phs_config%md5sum_phs_config = md5sum (u)
      close (u)
    end subroutine phs_config_compute_md5sum
```

Print an informative message after phase-space configuration.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure (phs_startup_message), deferred :: startup_message
    procedure :: base_startup_message => phs_startup_message
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_startup_message (phs_config, unit)
      class(phs_config_t), intent(in) :: phs_config
      integer, intent(in), optional :: unit
      write (msg_buffer, "(A,3(1x,I0,1x,A))") &
           "Phase space:", &
           phs_config%n_channel, "channels,", &
           phs_config%n_par, "dimensions"
      call msg_message (unit = unit)
    end subroutine phs_startup_message
```

This procedure should be implemented such that the phase-space configuration object allocates a phase-space instance of matching type.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure (phs_config_allocate_instance), nopass, deferred :: &
         allocate_instance
```

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
      subroutine phs_config_allocate_instance (phs)
        import
        class(phs_t), intent(inout), pointer :: phs
      end subroutine phs_config_allocate_instance
    end interface
```

11.2.4 Extract data

Return the number of MC input parameters.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: get_n_par => phs_config_get_n_par
```

⟨*PHS base: procedures*⟩+≡
```
    function phs_config_get_n_par (phs_config) result (n)
      class(phs_config_t), intent(in) :: phs_config
      integer :: n
      n = phs_config%n_par
    end function phs_config_get_n_par
```

Return dimensions (parameter indices) for which the phase-space dimension is flat, so integration and event generation can be simplified.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: get_flat_dimensions => phs_config_get_flat_dimensions
```

⟨*PHS base: procedures*⟩+≡
```
    function phs_config_get_flat_dimensions (phs_config) result (dim_flat)
      class(phs_config_t), intent(in) :: phs_config
      integer, dimension(:), allocatable :: dim_flat
      if (allocated (phs_config%dim_flat)) then
         allocate (dim_flat (size (phs_config%dim_flat)))
         dim_flat = phs_config%dim_flat
      else
         allocate (dim_flat (0))
      end if
    end function phs_config_get_flat_dimensions
```

Return the number of phase-space channels.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: get_n_channel => phs_config_get_n_channel
```

⟨*PHS base: procedures*⟩+≡
```
    function phs_config_get_n_channel (phs_config) result (n)
      class(phs_config_t), intent(in) :: phs_config
      integer :: n
      n = phs_config%n_channel
    end function phs_config_get_n_channel
```

Return the structure-function channel that corresponds to the phase-space channel `c`. If the channel array is not allocated (which happens if there is no structure function), return zero.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: get_sf_channel => phs_config_get_sf_channel
```

⟨*PHS base: procedures*⟩+≡
```
    function phs_config_get_sf_channel (phs_config, c) result (c_sf)
      class(phs_config_t), intent(in) :: phs_config
      integer, intent(in) :: c
      integer :: c_sf
      if (allocated (phs_config%channel)) then
         c_sf = phs_config%channel(c)%sf_channel
```

```
    else
       c_sf = 0
    end if
end function phs_config_get_sf_channel
```

Return the mass(es) of the incoming particle(s). We take the first flavor combination in the array, assuming that masses must be degenerate among flavors.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: get_masses_in => phs_config_get_masses_in
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_config_get_masses_in (phs_config, m)
      class(phs_config_t), intent(in) :: phs_config
      real(default), dimension(:), intent(out) :: m
      integer :: i
      do i = 1, phs_config%n_in
         m(i) = flavor_get_mass (phs_config%flv(i,1))
      end do
    end subroutine phs_config_get_masses_in
```

Return the MD5 sum of the configuration.

⟨*PHS base: phs config: TBP*⟩+≡
```
    procedure :: get_md5sum => phs_config_get_md5sum
```

⟨*PHS base: procedures*⟩+≡
```
    function phs_config_get_md5sum (phs_config) result (md5sum)
      class(phs_config_t), intent(in) :: phs_config
      character(32) :: md5sum
      md5sum = phs_config%md5sum_phs_config
    end function phs_config_get_md5sum
```

11.2.5 Phase-space point instance

The `phs_t` object holds the workspace for phase-space generation. In the base object, we have the MC input parameters `r` and the Jacobian factor `f`, for each channel, and the incoming and outgoing momenta.

Note: The `active_channel` array is not used yet, all elements are initialized with `.true.`. It should be touched by the integrator if it decides to drop irrelevant channels.

⟨*PHS base: public*⟩+≡
```
    public :: phs_t
```

⟨*PHS base: types*⟩+≡
```
    type, abstract :: phs_t
      class(phs_config_t), pointer :: config => null ()
      logical :: r_defined = .false.
      integer :: selected_channel = 0
      logical, dimension(:), allocatable :: active_channel
      real(default), dimension(:,:), allocatable :: r
      real(default), dimension(:), allocatable :: f
      real(default), dimension(:), allocatable :: m_in
      real(default), dimension(:), allocatable :: m_out
```

```
    real(default) :: flux = 0
    real(default) :: volume = 0
    type(lorentz_transformation_t) :: lt_cm_to_lab
    logical :: p_defined = .false.
    real(default) :: sqrts_hat = 0
    type(vector4_t), dimension(:), allocatable :: p
    logical :: q_defined = .false.
    type(vector4_t), dimension(:), allocatable :: q
  contains
⟨PHS base: phs: TBP⟩
  end type phs_t
```

Output. Since phase space may get complicated, we include a `verbose` option for the abstract `write` procedure.

⟨*PHS base: phs: TBP*⟩≡
```
    procedure (phs_write), deferred :: write
```

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
      subroutine phs_write (object, unit, verbose)
        import
        class(phs_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose
      end subroutine phs_write
    end interface
```

This procedure can be called to print the contents of the base type.

⟨*PHS base: phs: TBP*⟩+≡
```
    procedure :: base_write => phs_base_write
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine phs_base_write (object, unit)
      class(phs_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, c, i
      u = output_unit (unit)
      write (u, "(1x,A)", advance="no")  "Partonic phase space: parameters"
      if (object%r_defined) then
         write (u, *)
      else
         write (u, "(1x,A)")  "[undefined]"
      end if
      write (u, "(3x,A,999(1x," // FMT_19 // "))") "m_in    =", object%m_in
      write (u, "(3x,A,999(1x," // FMT_19 // "))") "m_out   =", object%m_out
      write (u, "(3x,A," // FMT_19 // ")")  "Flux   = ", object%flux
      write (u, "(3x,A," // FMT_19 // ")")  "Volume = ", object%volume
      if (allocated (object%f)) then
         do c = 1, size (object%r, 2)
            write (u, "(1x,A,I0,A)", advance="no")  "Channel #", c, ":"
            if (c == object%selected_channel) then
               write (u, "(1x,A)")  "[selected]"
            else
               write (u, *)
```

```
            end if
            write (u, "(3x,A)", advance="no")  "r ="
            do i = 1, size (object%r, 1)
               write (u, "(1x,F9.7)", advance="no")  object%r(i,c)
            end do
            write (u, *)
            write (u, "(3x,A,1x,ES13.7)")  "f =", object%f(c)
         end do
      end if
      write (u, "(1x,A)")  "Partonic phase space: momenta"
      if (object%p_defined) then
         write (u, "(3x,A," // FMT_19 // ")")  "sqrts  = ", object%sqrts_hat
      end if
      write (u, "(1x,A)", advance="no")  "Incoming:"
      if (object%p_defined) then
         write (u, *)
      else
         write (u, "(1x,A)")  "[undefined]"
      end if
      if (allocated (object%p)) then
         do i = 1, size (object%p)
            call vector4_write (object%p(i), u)
         end do
      end if
      write (u, "(1x,A)", advance="no")  "Outgoing:"
      if (object%q_defined) then
         write (u, *)
      else
         write (u, "(1x,A)")  "[undefined]"
      end if
      if (allocated (object%q)) then
         do i = 1, size (object%q)
            call vector4_write (object%q(i), u)
         end do
      end if
      if (object%p_defined .and. .not. object%config%cm_frame) then
         write (u, "(1x,A)")  "Transformation c.m -> lab frame"
         call lorentz_transformation_write (object%lt_cm_to_lab, u)
      end if
    end subroutine phs_base_write
```

Finalizer. The base type does not need it, but extensions may.

⟨*PHS base: phs: TBP*⟩+≡

```
    procedure (phs_final), deferred :: final
```

⟨*PHS base: interfaces*⟩+≡

```
    abstract interface
       subroutine phs_final (object)
         import
         class(phs_t), intent(inout) :: object
       end subroutine phs_final
    end interface
```

Initializer. Everything should be contained in the `process_data` configuration object, so we can require a universal interface.

⟨*PHS base: phs: TBP*⟩+≡
```
procedure (phs_init), deferred :: init
```

⟨*PHS base: interfaces*⟩+≡
```
abstract interface
   subroutine phs_init (phs, phs_config)
     import
     class(phs_t), intent(out) :: phs
     class(phs_config_t), intent(in), target :: phs_config
   end subroutine phs_init
end interface
```

The base version will just allocate the arrays. It should be called at the beginning of the implementation of `phs_init`.

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: base_init => phs_base_init
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_base_init (phs, phs_config)
  class(phs_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config
  phs%config => phs_config
  allocate (phs%active_channel (phs%config%n_channel))
  phs%active_channel = .true.
  allocate (phs%r (phs%config%n_par, phs%config%n_channel));  phs%r = 0
  allocate (phs%f (phs%config%n_channel));                    phs%f = 0
  allocate (phs%p (phs%config%n_in))
  allocate (phs%q (phs%config%n_out))
  allocate (phs%m_in  (phs%config%n_in), &
       source = flavor_get_mass (phs_config%flv(:phs_config%n_in, 1)))
  allocate (phs%m_out (phs%config%n_out), &
       source = flavor_get_mass (phs_config%flv(phs_config%n_in+1:, 1)))
  call phs%compute_flux ()
end subroutine phs_base_init
```

Manually select a channel.

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: select_channel => phs_base_select_channel
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_base_select_channel (phs, channel)
  class(phs_t), intent(inout) :: phs
  integer, intent(in), optional :: channel
  if (present (channel)) then
     phs%selected_channel = channel
  else
     phs%selected_channel = 0
  end if
end subroutine phs_base_select_channel
```

Set incoming momenta. Assume that array shapes match. If requested, compute the Lorentz transformation from the c.m. to the lab frame and apply that transformation to the incoming momenta.

In the c.m. frame, the sum of three-momenta is zero. In a scattering process, the z axis is the direction of the first beam, the second beam is along the negative z axis. The transformation from the c.m. to the lab frame is a rotation from the z axis to the boost axis followed by a boost, such that the c.m. momenta are transformed into the lab-frame momenta. In a decay process, we just boost along the flight direction, without rotation.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: set_incoming_momenta => phs_set_incoming_momenta
```

⟨*PHS base: procedures*⟩+≡
```
   subroutine phs_set_incoming_momenta (phs, p)
     class(phs_t), intent(inout) :: phs
     type(vector4_t), dimension(:), intent(in) :: p
     type(vector4_t) :: p0, p1
     type(lorentz_transformation_t) :: lt0
     phs%p = p
     if (phs%config%cm_frame) then
        phs%sqrts_hat = phs%config%sqrts
        phs%p = p
     else
        p0 = sum (p)
        if (phs%config%sqrts_fixed) then
           phs%sqrts_hat = phs%config%sqrts
        else
           phs%sqrts_hat = p0 ** 1
        end if
        lt0 = boost (p0, phs%sqrts_hat)
        select case (phs%config%n_in)
        case (1)
           phs%lt_cm_to_lab = lt0
        case (2)
           p1 = inverse (lt0) * p(1)
           phs%lt_cm_to_lab = lt0 * rotation_to_2nd (3, space_part (p1))
        end select
        phs%p = inverse (phs%lt_cm_to_lab) * p
     end if
     phs%p_defined = .true.
   end subroutine phs_set_incoming_momenta
```

Set outgoing momenta. Assume that array shapes match. The incoming momenta must be known, so can apply the Lorentz transformation from c.m. to lab (inverse) to the momenta.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: set_outgoing_momenta => phs_set_outgoing_momenta
```

⟨*PHS base: procedures*⟩+≡
```
   subroutine phs_set_outgoing_momenta (phs, q)
     class(phs_t), intent(inout) :: phs
     type(vector4_t), dimension(:), intent(in) :: q
     if (phs%p_defined) then
```

```
      if (phs%config%cm_frame) then
         phs%q = q
      else
         phs%q = inverse (phs%lt_cm_to_lab) * q
      end if
      phs%q_defined = .true.
   end if
end subroutine phs_set_outgoing_momenta
```

Return outgoing momenta. Apply the c.m. to lab transformation if necessary.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: get_outgoing_momenta => phs_get_outgoing_momenta
```

⟨*PHS base: procedures*⟩+≡
```
   subroutine phs_get_outgoing_momenta (phs, q)
     class(phs_t), intent(in) :: phs
     type(vector4_t), dimension(:), intent(out) :: q
     if (phs%p_defined .and. phs%q_defined) then
        if (phs%config%cm_frame) then
           q = phs%q
        else
           q = phs%lt_cm_to_lab * phs%q
        end if
     else
        q = vector4_null
     end if
   end subroutine phs_get_outgoing_momenta
```

Return the input parameter array for a channel.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: get_mcpar => phs_get_mcpar
```

⟨*PHS base: procedures*⟩+≡
```
   subroutine phs_get_mcpar (phs, c, r)
     class(phs_t), intent(in) :: phs
     integer, intent(in) :: c
     real(default), dimension(:), intent(out) :: r
     if (phs%r_defined) then
        r = phs%r(:,c)
     else
        r = 0
     end if
   end subroutine phs_get_mcpar
```

Return the Jacobian factor for a channel.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: get_f => phs_get_f
```

⟨*PHS base: procedures*⟩+≡
```
   function phs_get_f (phs, c) result (f)
     class(phs_t), intent(in) :: phs
     integer, intent(in) :: c
     real(default) :: f
     if (phs%r_defined) then
```

```
        f = phs%f(c)
      else
         f = 0
      end if
   end function phs_get_f
```

Return the overall factor, which is the product of the flux factor for the incoming partons and the phase-space volume for the outgoing partons.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: get_overall_factor => phs_get_overall_factor
```

⟨*PHS base: procedures*⟩+≡
```
   function phs_get_overall_factor (phs) result (f)
     class(phs_t), intent(in) :: phs
     real(default) :: f
     f = phs%flux * phs%volume
   end function phs_get_overall_factor
```

Compute flux factor. We do this during initialization (when the incoming momenta `p` are undefined), unless `sqrts` is variable. We do this again once for each phase-space point, but then we skip the calculation if `sqrts` is fixed.

There are three different flux factors.

1. For a decaying massive particle, the factor is

$$f = (2\pi)^4/(2M) \tag{11.1}$$

2. For a $2 \to n$ scattering process with $n > 1$, the factor is

$$f = (2\pi)^4/(2\sqrt{\lambda}) \tag{11.2}$$

where for massless incoming particles, $\sqrt{\lambda} = s$.

3. For a $2 \to 1$ on-shell production process, the factor includes an extra $1/(2\pi)^3$ factor and a $1/m^2$ factor from the phase-space delta function $\delta(x_1 x_2 - m^2/s)$, which originate from the one-particle phase space that we integrate out.

$$f = 2\pi/(2sm^2) \tag{11.3}$$

The delta function is handled by the structure-function parameterization.

⟨*PHS base: phs: TBP*⟩+≡
```
   procedure :: compute_flux => phs_compute_flux
```

⟨*PHS base: procedures*⟩+≡
```
   subroutine phs_compute_flux (phs)
     class(phs_t), intent(inout) :: phs
     real(default) :: s_hat, lda
     select case (phs%config%n_in)
     case (1)
        if (.not. phs%p_defined) then
           phs%flux = twopi4 / (2 * phs%m_in(1))
        end if
     case (2)
```

```
            if (phs%p_defined) then
               if (phs%config%sqrts_fixed) then
                  return
               else
                  s_hat = sum (phs%p) ** 2
               end if
            else
               if (phs%config%sqrts_fixed) then
                  s_hat = phs%config%sqrts ** 2
               else
                  return
               end if
            end if
            select case (phs%config%n_out)
            case (2:)
               lda = lambda (s_hat, phs%m_in(1) ** 2, phs%m_in(2) ** 2)
               if (lda > 0) then
                  phs%flux = conv * twopi4 / (2 * sqrt (lda))
               else
                  phs%flux = 0
               end if
            case (1)
               phs%flux = conv * twopi &
                     / (2 * phs%config%sqrts ** 2 * phs%m_out(1) ** 2)
            case default
               phs%flux = 0
            end select
         end select
    end subroutine phs_compute_flux
```

Evaluate the phase-space point for a particular channel and compute momenta, Jacobian, and phase-space volume. This is, of course, deferred to the implementation.

⟨*PHS base: phs: TBP*⟩+≡
```
    procedure (phs_evaluate_selected_channel), deferred :: &
         evaluate_selected_channel
```

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
       subroutine phs_evaluate_selected_channel (phs, c_in, r_in)
         import
         class(phs_t), intent(inout) :: phs
         integer, intent(in) :: c_in
         real(default), dimension(:), intent(in) :: r_in
       end subroutine phs_evaluate_selected_channel
    end interface
```

Compute the inverse mappings to completely fill the `r` and `f` arrays, for the non-selected channels.

⟨*PHS base: phs: TBP*⟩+≡
```
    procedure (phs_evaluate_other_channels), deferred :: &
         evaluate_other_channels
```

1376

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
       subroutine phs_evaluate_other_channels (phs, c_in)
          import
          class(phs_t), intent(inout) :: phs
          integer, intent(in) :: c_in
       end subroutine phs_evaluate_other_channels
    end interface
```

Inverse evaluation. If all momenta are known, we compute the inverse mappings to fill the r and f arrays.

⟨*PHS base: phs: TBP*⟩+≡
```
    procedure (phs_inverse), deferred :: inverse
```

⟨*PHS base: interfaces*⟩+≡
```
    abstract interface
       subroutine phs_inverse (phs)
          import
          class(phs_t), intent(inout) :: phs
       end subroutine phs_inverse
    end interface
```

11.2.6   Auxiliary stuff

The `pacify` subroutine, which is provided by the Lorentz module, has the purpose of setting numbers to zero which are (by comparing with a `tolerance` parameter) considered equivalent with zero. This is useful for numerical checks.

⟨*PHS base: public*⟩+≡
```
    public :: pacify
```

⟨*PHS base: interfaces*⟩+≡
```
    interface pacify
       module procedure pacify_phs
    end interface pacify
```

⟨*PHS base: procedures*⟩+≡
```
    subroutine pacify_phs (phs)
      class(phs_t), intent(inout) :: phs
      if (phs%p_defined) then
         call pacify (phs%p, 30 * epsilon (1._default) * phs%config%sqrts)
         call pacify (phs%lt_cm_to_lab, 30 * epsilon (1._default))
      end if
      if (phs%q_defined) then
         call pacify (phs%q, 30 * epsilon (1._default) * phs%config%sqrts)
      end if
    end subroutine pacify_phs
```

11.2.7   Unit tests

⟨*PHS base: public*⟩+≡
```
    public :: phs_base_test
```

⟨PHS base: tests⟩≡
```
subroutine phs_base_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨PHS base: execute tests⟩
end subroutine phs_base_test
```

Test process data

We provide a procedure that initializes a test case for the process constants. This set of process data contains just the minimal contents that we need for the phase space. The rest is left uninitialized.

⟨PHS base: public⟩+≡
```
public :: init_test_process_data
```

⟨PHS base: tests⟩+≡
```
subroutine init_test_process_data (id, data)
  type(process_constants_t), intent(out) :: data
  type(string_t), intent(in), optional :: id
  if (present (id)) then
     data%id = id
  else
     data%id = "testproc"
  end if
  data%model_name = "Test"
  data%n_in = 2
  data%n_out = 2
  data%n_flv = 1
  allocate (data%flv_state (data%n_in + data%n_out, data%n_flv))
  data%flv_state = 25
end subroutine init_test_process_data
```

This is the variant for a decay process.

⟨PHS base: public⟩+≡
```
public :: init_test_decay_data
```

⟨PHS base: tests⟩+≡
```
subroutine init_test_decay_data (id, data)
  type(process_constants_t), intent(out) :: data
  type(string_t), intent(in), optional :: id
  if (present (id)) then
     data%id = id
  else
     data%id = "testproc"
  end if
  data%model_name = "Test"
  data%n_in = 1
  data%n_out = 2
  data%n_flv = 1
  allocate (data%flv_state (data%n_in + data%n_out, data%n_flv))
  data%flv_state(:,1) = [25, 6, -6]
end subroutine init_test_decay_data
```

Test kinematics configuration

This is a trivial implementation of the `phs_config_t` configuration object.

⟨*PHS base: public*⟩+≡
```
    public :: phs_test_config_t
```

⟨*PHS base: test types*⟩≡
```
    type, extends (phs_config_t) :: phs_test_config_t
      logical :: create_equivalences = .false.
     contains
      procedure :: final => phs_test_config_final
      procedure :: write => phs_test_config_write
      procedure :: configure => phs_test_config_configure
      procedure :: startup_message => phs_test_config_startup_message
      procedure, nopass :: allocate_instance => phs_test_config_allocate_instance
    end type phs_test_config_t
```

The finalizer is empty.

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_test_config_final (object)
      class(phs_test_config_t), intent(inout) :: object
    end subroutine phs_test_config_final
```

The `cm_frame` parameter is not tested here; we defer this to the `phs_single` implementation.

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_test_config_write (object, unit)
      class(phs_test_config_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "Partonic phase-space configuration:"
      call object%base_write (unit)
    end subroutine phs_test_config_write

    subroutine phs_test_config_configure (phs_config, sqrts, &
        sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch)
      class(phs_test_config_t), intent(inout) :: phs_config
      real(default), intent(in) :: sqrts
      logical, intent(in), optional :: sqrts_fixed
      logical, intent(in), optional :: cm_frame
      logical, intent(in), optional :: azimuthal_dependence
      logical, intent(in), optional :: rebuild
      logical, intent(in), optional :: ignore_mismatch
      phs_config%n_channel = 2
      phs_config%n_par = 2
      phs_config%sqrts = sqrts
      if (present (sqrts_fixed)) then
         phs_config%sqrts_fixed = sqrts_fixed
      end if
      if (present (cm_frame)) then
         phs_config%cm_frame = cm_frame
      end if
```

1379

```
      if (present (azimuthal_dependence)) then
         phs_config%azimuthal_dependence = azimuthal_dependence
      end if
      if (allocated (phs_config%channel))  deallocate (phs_config%channel)
      allocate (phs_config%channel (phs_config%n_channel))
      if (phs_config%create_equivalences) then
         call setup_test_equivalences (phs_config)
         call setup_test_channel_props (phs_config)
      end if
      call phs_config%compute_md5sum ()
    end subroutine phs_test_config_configure
```

If requested, we make up an arbitrary set of equivalences.

⟨*PHS base: tests*⟩+≡
```
    subroutine setup_test_equivalences (phs_config)
      class(phs_test_config_t), intent(inout) :: phs_config
      integer :: i
      associate (channel => phs_config%channel(1))
        allocate (channel%eq (2))
        do i = 1, size (channel%eq)
           call channel%eq(i)%init (phs_config%n_par)
        end do
        associate (eq => channel%eq(1))
          eq%c = 1;  eq%perm = [1, 2];  eq%mode = [EQ_IDENTITY, EQ_SYMMETRIC]
        end associate
        associate (eq => channel%eq(2))
          eq%c = 2;  eq%perm = [2, 1];  eq%mode = [EQ_INVARIANT, EQ_IDENTITY]
        end associate
      end associate
    end subroutine setup_test_equivalences
```

Ditto, for channel properties.

⟨*PHS base: tests*⟩+≡
```
    subroutine setup_test_channel_props (phs_config)
      class(phs_test_config_t), intent(inout) :: phs_config
      associate (channel => phs_config%channel(2))
        call channel%set_resonant (140._default, 3.1415_default)
      end associate
    end subroutine setup_test_channel_props
```

Startup message

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_test_config_startup_message (phs_config, unit)
      class(phs_test_config_t), intent(in) :: phs_config
      integer, intent(in), optional :: unit
      call phs_config%base_startup_message (unit)
      write (msg_buffer, "(A)") "Phase space: Test"
      call msg_message (unit = unit)
    end subroutine phs_test_config_startup_message
```

The instance type that matches `phs_test_config_t` is `phs_test_t`.

⟨*PHS base: tests*⟩+≡

```
    subroutine phs_test_config_allocate_instance (phs)
      class(phs_t), intent(inout), pointer :: phs
      allocate (phs_test_t :: phs)
    end subroutine phs_test_config_allocate_instance
```

Test kinematics implementation

This implementation of kinematics generates a simple two-particle configuration from the incoming momenta. The incoming momenta must be in the c.m. system, all masses equal.

There are two channels: one generates $\cos\theta$ and ϕ uniformly, in the other channel we map the r_1 parameter which belongs to $\cos\theta$.

We should store the mass parameter that we need.

⟨*PHS base: public*⟩+≡
```
    public :: phs_test_t
```

⟨*PHS base: test types*⟩+≡
```
    type, extends (phs_t) :: phs_test_t
      real(default) :: m = 0
      real(default), dimension(:), allocatable :: x
     contains
```
    ⟨*PHS base: phs test: TBP*⟩
```
    end type phs_test_t
```

Output. The specific data are displayed only if `verbose` is set.

⟨*PHS base: phs test: TBP*⟩≡
```
    procedure :: write => phs_test_write
```

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_test_write (object, unit, verbose)
      class(phs_test_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      logical :: verb
      u = output_unit (unit)
      verb = .false.;  if (present (verbose))  verb = verbose
      if (verb) then
         write (u, "(1x,A)")  "Partonic phase space: data"
         write (u, "(3x,A," // FMT_19 // ")")  "m = ", object%m
      end if
      call object%base_write (u)
    end subroutine phs_test_write
```

The finalizer is empty.

⟨*PHS base: phs test: TBP*⟩+≡
```
    procedure :: final => phs_test_final
```

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_test_final (object)
      class(phs_test_t), intent(inout) :: object
    end subroutine phs_test_final
```

Initialization: set the mass value.

⟨PHS base: phs test: TBP⟩+≡
```
    procedure :: init => phs_test_init
```

⟨PHS base: tests⟩+≡
```
    subroutine phs_test_init (phs, phs_config)
      class(phs_test_t), intent(out) :: phs
      class(phs_config_t), intent(in), target :: phs_config
      call phs%base_init (phs_config)
      phs%m = flavor_get_mass (phs%config%flv(1,1))
      allocate (phs%x (phs_config%n_par), source = 0._default)
    end subroutine phs_test_init
```

Evaluation. In channel 1, we uniformly generate $\cos\theta$ and ϕ, with Jacobian normalized to one. In channel 2, we prepend a mapping $r_1 \rightarrow r_1^{(}1/3)$ with Jacobian $f = 3r_1^2$.

The component x is allocated in the first subroutine, used and deallocated in the second one.

⟨PHS base: phs test: TBP⟩+≡
```
    procedure :: evaluate_selected_channel => phs_test_evaluate_selected_channel
    procedure :: evaluate_other_channels => phs_test_evaluate_other_channels
```

⟨PHS base: tests⟩+≡
```
    subroutine phs_test_evaluate_selected_channel (phs, c_in, r_in)
      class(phs_test_t), intent(inout) :: phs
      integer, intent(in) :: c_in
      real(default), intent(in), dimension(:) :: r_in
      if (phs%p_defined) then
         call phs%select_channel (c_in)
         phs%r(:,c_in) = r_in
         select case (c_in)
         case (1)
            phs%x = r_in
         case (2)
            phs%x(1) = r_in(1) ** (1 / 3._default)
            phs%x(2) = r_in(2)
         end select
         call compute_kinematics_solid_angle (phs%p, phs%q, phs%x)
         phs%volume = 1
         phs%q_defined = .true.
      end if
    end subroutine phs_test_evaluate_selected_channel

    subroutine phs_test_evaluate_other_channels (phs, c_in)
      class(phs_test_t), intent(inout) :: phs
      integer, intent(in) :: c_in
      integer :: c, n_channel
      if (phs%p_defined) then
         n_channel = phs%config%n_channel
         do c = 1, n_channel
            if (c /= c_in) then
               call inverse_kinematics_solid_angle (phs%p, phs%q, phs%x)
               select case (c)
               case (1)
```

```
                phs%r(:,c) = phs%x
             case (2)
                phs%r(1,c) = phs%x(1) ** 3
                phs%r(2,c) = phs%x(2)
             end select
          end if
       end do
       phs%f(1) = 1
       if (phs%r(1,2) /= 0) then
          phs%f(2) = 1 / (3 * phs%r(1,2) ** (2/3._default))
       else
          phs%f(2) = 0
       end if
       phs%r_defined = .true.
    end if
  end subroutine phs_test_evaluate_other_channels
```

Inverse evaluation.

⟨*PHS base: phs test: TBP*⟩+≡
```
    procedure :: inverse => phs_test_inverse
```

⟨*PHS base: tests*⟩+≡
```
  subroutine phs_test_inverse (phs)
    class(phs_test_t), intent(inout) :: phs
    integer :: c, n_channel
    real(default), dimension(:), allocatable :: x
    if (phs%p_defined .and. phs%q_defined) then
       call phs%select_channel ()
       n_channel = phs%config%n_channel
       allocate (x (phs%config%n_par))
       do c = 1, n_channel
          call inverse_kinematics_solid_angle (phs%p, phs%q, x)
          select case (c)
          case (1)
             phs%r(:,c) = x
          case (2)
             phs%r(1,c) = x(1) ** 3
             phs%r(2,c) = x(2)
          end select
       end do
       phs%f(1) = 1
       if (phs%r(1,2) /= 0) then
          phs%f(2) = 1 / (3 * phs%r(1,2) ** (2/3._default))
       else
          phs%f(2) = 0
       end if
       phs%volume = 1
       phs%r_defined = .true.
    end if
  end subroutine phs_test_inverse
```

1383

Uniform angular distribution

These procedures implement the uniform angular distribution, generated from two parameters x_1 and x_2:

$$\cos\theta = 1 - 2x_1, \qquad \phi = 2\pi x_2 \tag{11.4}$$

We generate a rotation (Lorentz transformation) which rotates the positive z axis into this point on the unit sphere. This rotation is applied to the **p** momenta, which are assumed to be back-to-back, on-shell, and with the correct mass.

We do not compute a Jacobian (constant). The uniform distribution is assumed to be normalized.

⟨*PHS base: public*⟩+≡
```
public :: compute_kinematics_solid_angle
```

⟨*PHS base: tests*⟩+≡
```
subroutine compute_kinematics_solid_angle (p, q, x)
  type(vector4_t), dimension(2), intent(in) :: p
  type(vector4_t), dimension(2), intent(out) :: q
  real(default), dimension(2), intent(in) :: x
  real(default) :: ct, st, phi
  type(lorentz_transformation_t) :: rot
  integer :: i
  ct = 1 - 2*x(1)
  st = sqrt (1 - ct**2)
  phi = twopi * x(2)
  rot = rotation (phi, 3) * rotation (ct, st, 2)
  do i = 1, 2
     q(i) = rot * p(i)
  end do
end subroutine compute_kinematics_solid_angle
```

This is the inverse transformation. We assume that the outgoing momenta are rotated versions of the incoming momenta, back-to-back. Thus, we determine the angles from $q(1)$ alone. **p** is unused.

⟨*PHS base: public*⟩+≡
```
public :: inverse_kinematics_solid_angle
```

⟨*PHS base: tests*⟩+≡
```
subroutine inverse_kinematics_solid_angle (p, q, x)
  type(vector4_t), dimension(:), intent(in) :: p
  type(vector4_t), dimension(2), intent(in) :: q
  real(default), dimension(2), intent(out) :: x
  real(default) :: ct, phi
  ct = polar_angle_ct (q(1))
  phi = azimuthal_angle (q(1))
  x(1) = (1 - ct) / 2
  x(2) = phi / twopi
end subroutine inverse_kinematics_solid_angle
```

Phase-space configuration data

Construct and display a test phase-space configuration object.

⟨*PHS base: execute tests*⟩≡

```
        call test (phs_base_1, "phs_base_1", &
             "phase-space configuration", &
             u, results)
```

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_base_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_constants_t) :: process_data
      class(phs_config_t), allocatable :: phs_data

      write (u, "(A)")  "* Test output: phs_base_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
           &test phase-space configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)

      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_base_1"), process_data)

      allocate (phs_test_config_t :: phs_data)
      call phs_data%init (process_data, model)

      call phs_data%write (u)

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_base_1"

    end subroutine phs_base_1
```

Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

⟨*PHS base: execute tests*⟩+≡
```
        call test (phs_base_2, "phs_base_2", &
             "phase-space evaluation", &
             u, results)
```

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_base_2 (u)
      integer, intent(in) :: u
```

```
type(os_data_t) :: os_data
type(model_list_t) :: model_list
type(model_t), pointer :: model
type(flavor_t) :: flv
type(process_constants_t) :: process_data
real(default) :: sqrts, E
class(phs_config_t), allocatable, target :: phs_data
class(phs_t), pointer :: phs => null ()
type(vector4_t), dimension(2) :: p, q

write (u, "(A)")  "* Test output: phs_base_2"
write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
call flavor_init (flv, 25, model)

write (u, "(A)")  "* Initialize a process and a matching &
     &phase-space configuration"
write (u, "(A)")

call init_test_process_data (var_str ("phs_base_2"), process_data)

allocate (phs_test_config_t :: phs_data)
call phs_data%init (process_data, model)

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
select type (phs)
type is (phs_test_t)
   call phs%init (phs_data)
end select

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv)**2), 3)
```

```
call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point in channel 1 &
     &for x = 0.5, 0.125"
write (u, "(A)")

call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
call phs%evaluate_other_channels (1)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point in channel 2 &
     &for x = 0.125, 0.125"
write (u, "(A)")

call phs%evaluate_selected_channel (2, [0.125_default, 0.125_default])
call phs%evaluate_other_channels (2)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

sqrts = 1000._default
select type (phs_data)
type is (phs_test_config_t)
   call phs_data%configure (sqrts)
end select

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call phs%write (u)

call phs%final ()
deallocate (phs)

call phs_data%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_base_2"
```

```
    end subroutine phs_base_2
```

Phase-space equivalences

Construct a test phase-space configuration which contains channel equivalences.

⟨*PHS base: execute tests*⟩+≡
```
    call test (phs_base_3, "phs_base_3", &
         "channel equivalences", &
         u, results)
```

⟨*PHS base: tests*⟩+≡
```
    subroutine phs_base_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_constants_t) :: process_data
      class(phs_config_t), allocatable :: phs_data

      write (u, "(A)")  "* Test output: phs_base_3"
      write (u, "(A)")  "*   Purpose: construct phase-space configuration data &
           &with equivalences"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)

      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_base_3"), process_data)

      allocate (phs_test_config_t :: phs_data)
      call phs_data%init (process_data, model)
      select type (phs_data)
      type is (phs_test_config_t)
         phs_data%create_equivalences = .true.
      end select

      call phs_data%configure (1000._default)
      call phs_data%write (u)

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_base_3"

    end subroutine phs_base_3
```

MD5 sum checks

Construct a test phase-space configuration, compute and compare MD5 sums.

⟨*PHS base: execute tests*⟩+≡
```
call test (phs_base_4, "phs_base_4", &
     "MD5 sum", &
     u, results)
```

⟨*PHS base: tests*⟩+≡
```
subroutine phs_base_4 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(process_constants_t) :: process_data
  class(phs_config_t), allocatable :: phs_data
  type(var_list_t), pointer :: var_list

  write (u, "(A)")  "* Test output: phs_base_4"
  write (u, "(A)")  "*   Purpose: compute and compare MD5 sums"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)

  write (u, "(A)")  "* Model parameters"
  write (u, "(A)")

  var_list => model_get_var_list_ptr (model)
  call var_list_write (var_list, u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize a process and a matching &
       &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_base_4"), process_data)
  process_data%md5sum = "test_process_data_m6sum_12345678"

  allocate (phs_test_config_t :: phs_data)
  call phs_data%init (process_data, model)

  call phs_data%compute_md5sum ()
  call phs_data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Modify model parameter"
  write (u, "(A)")

  call var_list_set_real (var_list, var_str ("ms"), 100._default, &
```

1389

```
            is_known=.true.)
       call var_list_write (var_list, u)

       write (u, "(A)")
       write (u, "(A)")   "* PHS configuration"
       write (u, "(A)")

       call phs_data%compute_md5sum ()
       call phs_data%write (u)

       call phs_data%final ()
       call model_list%final ()
       call syntax_model_file_final ()

       write (u, "(A)")
       write (u, "(A)")   "* Test output end: phs_base_4"

     end subroutine phs_base_4
```

Phase-space channel collection

Set up an array of various phase-space channels and collect them in a list.

⟨*PHS base: execute tests*⟩+≡
```
     call test (phs_base_5, "phs_base_5", &
          "channel collection", &
          u, results)
```

⟨*PHS base: tests*⟩+≡
```
     subroutine phs_base_5 (u)
       integer, intent(in) :: u
       type(phs_channel_t), dimension(:), allocatable :: channel
       type(phs_channel_collection_t) :: coll
       integer :: i, n

       write (u, "(A)")   "* Test output: phs_base_5"
       write (u, "(A)")   "*   Purpose: collect channel properties"
       write (u, "(A)")

       write (u, "(A)")   "* Set up an array of channels"
       write (u, "(A)")

       n = 6

       allocate (channel (n))
       call channel(2)%set_resonant (75._default, 3._default)
       call channel(4)%set_resonant (130._default, 1._default)
       call channel(5)%set_resonant (75._default, 3._default)
       call channel(6)%set_on_shell (33._default)

       do i = 1, n
          write (u, "(1x,I0)", advance="no")  i
          call channel(i)%write (u)
       end do
```

1390

```
  write (u, "(A)")
  write (u, "(A)")  "* Collect distinct properties"
  write (u, "(A)")

  do i = 1, n
     call coll%push (channel(i))
  end do

  write (u, "(1x,A,I0)")  "n = ", coll%get_n ()
  write (u, "(A)")

  call coll%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Channel array with collection index assigned"
  write (u, "(A)")

  do i = 1, n
     write (u, "(1x,I0)", advance="no")  i
     call channel(i)%write (u)
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call coll%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_base_5"

end subroutine phs_base_5
```

11.3 Single-particle phase space

This module implements the phase space for a single particle, i.e., the solid angle, in a straightforward parameterization with a single channel. The phase-space implementation may be used either for $1 \rightarrow 2$ decays or for $2 \rightarrow 2$ scattering processes, so the number of incoming particles is the only free parameter in the configuration. In the latter case, we should restrict its use to non-resonant s-channel processes, because there is no mapping of the scattering angle.

(We might extend this later to account for generic $2 \rightarrow 2$ situations, e.g., account for a Coulomb singularity or detect an s-channel resonance structure that requires matching structure-function mappings.)

This is derived from the `phs_test` implementation in the `phs_base` module above, even more simplified, but intended for actual use.

⟨`phs_single.f90`⟩≡
  ⟨*File header*⟩

  ```
  module phs_single
  ```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use unit_tests
    use os_interface
    use constants !NODEP!
    use lorentz !NODEP!
    use variables
    use models
    use flavors
    use process_constants
    use sf_mappings
    use sf_base
    use phs_base
  ```

  ⟨*Standard module head*⟩

  ⟨*PHS single: public*⟩

  ⟨*PHS single: types*⟩

  ```
  contains
  ```

  ⟨*PHS single: procedures*⟩

  ⟨*PHS single: tests*⟩

  ```
  end module phs_single
  ```

11.3.1 Configuration

⟨*PHS single: public*⟩≡
  ```
    public :: phs_single_config_t
  ```

⟨*PHS single: types*⟩≡
```
type, extends (phs_config_t) :: phs_single_config_t
 contains
 ⟨PHS single: phs single config: TBP⟩
end type phs_single_config_t
```

The finalizer is empty.

⟨*PHS single: phs single config: TBP*⟩≡
```
procedure :: final => phs_single_config_final
```

⟨*PHS single: procedures*⟩≡
```
subroutine phs_single_config_final (object)
  class(phs_single_config_t), intent(inout) :: object
end subroutine phs_single_config_final
```

Output.

⟨*PHS single: phs single config: TBP*⟩+≡
```
procedure :: write => phs_single_config_write
```

⟨*PHS single: procedures*⟩+≡
```
subroutine phs_single_config_write (object, unit)
  class(phs_single_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)")  "Partonic phase-space configuration (single-particle):"
  call object%base_write (unit)
end subroutine phs_single_config_write
```

Configuration: there is only one channel and two parameters. The second parameter is the azimuthal angle, which may be a flat dimension.

⟨*PHS single: phs single config: TBP*⟩+≡
```
procedure :: configure => phs_single_config_configure
```

⟨*PHS single: procedures*⟩+≡
```
subroutine phs_single_config_configure (phs_config, sqrts, &
    sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch)
  class(phs_single_config_t), intent(inout) :: phs_config
  real(default), intent(in) :: sqrts
  logical, intent(in), optional :: sqrts_fixed
  logical, intent(in), optional :: cm_frame
  logical, intent(in), optional :: azimuthal_dependence
  logical, intent(in), optional :: rebuild
  logical, intent(in), optional :: ignore_mismatch
  if (phs_config%n_out == 2) then
     phs_config%n_channel = 1
     phs_config%n_par = 2
     phs_config%sqrts = sqrts
     if (present (sqrts_fixed))  phs_config%sqrts_fixed = sqrts_fixed
     if (present (cm_frame))  phs_config%cm_frame = cm_frame
     if (present (azimuthal_dependence)) then
        phs_config%azimuthal_dependence = azimuthal_dependence
        if (.not. azimuthal_dependence) then
```

1393

```
              allocate (phs_config%dim_flat (1))
              phs_config%dim_flat(1) = 2
          end if
       end if
       if (allocated (phs_config%channel))  deallocate (phs_config%channel)
       allocate (phs_config%channel (1))
       call phs_config%compute_md5sum ()
    else
       call msg_fatal ("Single-particle phase space requires n_out = 2")
    end if
  end subroutine phs_single_config_configure
```

Startup message, after configuration is complete.

⟨*PHS single: phs single config: TBP*⟩+≡
```
    procedure :: startup_message => phs_single_config_startup_message
```

⟨*PHS single: procedures*⟩+≡
```
    subroutine phs_single_config_startup_message (phs_config, unit)
      class(phs_single_config_t), intent(in) :: phs_config
      integer, intent(in), optional :: unit
      call phs_config%base_startup_message (unit)
      write (msg_buffer, "(A,2(1x,I0,1x,A))") &
           "Phase space: single-particle"
      call msg_message (unit = unit)
    end subroutine phs_single_config_startup_message
```

Allocate an instance: the actual phase-space object.

⟨*PHS single: phs single config: TBP*⟩+≡
```
    procedure, nopass :: allocate_instance => phs_single_config_allocate_instance
```

⟨*PHS single: procedures*⟩+≡
```
    subroutine phs_single_config_allocate_instance (phs)
      class(phs_t), intent(inout), pointer :: phs
      allocate (phs_single_t :: phs)
    end subroutine phs_single_config_allocate_instance
```

11.3.2   Kinematics implementation

We generate $\cos\theta$ and ϕ uniformly, covering the solid angle.

Note: The incoming momenta must be in the c.m. system.

⟨*PHS single: public*⟩+≡
```
    public :: phs_single_t
```

⟨*PHS single: types*⟩+≡
```
    type, extends (phs_t) :: phs_single_t
     contains
     ⟨PHS single: phs single: TBP⟩
    end type phs_single_t
```

Output. The verbose setting is irrelevant, we just display the contents of the base object.

⟨*PHS single: phs single: TBP*⟩≡
```
    procedure :: write => phs_single_write
```

⟨*PHS single: procedures*⟩+≡
```
    subroutine phs_single_write (object, unit, verbose)
      class(phs_single_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      u = output_unit (unit)
      call object%base_write (u)
    end subroutine phs_single_write
```

The finalizer is empty.

⟨*PHS single: phs single: TBP*⟩+≡
```
    procedure :: final => phs_single_final
```

⟨*PHS single: procedures*⟩+≡
```
    subroutine phs_single_final (object)
      class(phs_single_t), intent(inout) :: object
    end subroutine phs_single_final
```

Initialization. We allocate arrays (`base_init`) and adjust the phase-space volume. The massless two-particle phase space volume is

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.55294034614 \times 10^{-5} \tag{11.5}$$

For a decay with nonvanishing masses (m_3, m_4), there is a correction factor

$$\Phi_2(m)/\Phi_2(0) = \frac{1}{\hat{s}} \lambda^{1/2}(\hat{s}, m_3^2, m_4^2). \tag{11.6}$$

For a scattering process with nonvanishing masses, the correction factor is

$$\Phi_2(m)/\Phi_2(0) = \frac{1}{\hat{s}^2} \lambda^{1/2}(\hat{s}, m_1^2, m_2^2)\, \lambda^{1/2}(\hat{s}, m_3^2, m_4^2). \tag{11.7}$$

If the energy is fixed, this is constant. Otherwise, we have to account for varying $\hat{s}$.

⟨*PHS single: phs single: TBP*⟩+≡
```
    procedure :: init => phs_single_init
```

⟨*PHS single: procedures*⟩+≡
```
    subroutine phs_single_init (phs, phs_config)
      class(phs_single_t), intent(out) :: phs
      class(phs_config_t), intent(in), target :: phs_config
      call phs%base_init (phs_config)
      phs%volume = 1 / (4 * twopi5)
      call phs%compute_factor ()
    end subroutine phs_single_init
```

Compute the correction factor for nonzero masses. We do this during initialization (when the incoming momenta `p` are undefined), unless `sqrts` is variable. We do this again once for each phase-space point, but then we skip the calculation if `sqrts` is fixed.

⟨*PHS single: phs single: TBP*⟩+≡
```
    procedure :: compute_factor => phs_single_compute_factor
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_compute_factor (phs)
  class(phs_single_t), intent(inout) :: phs
  real(default) :: s_hat
  select case (phs%config%n_in)
  case (1)
     if (.not. phs%p_defined) then
        if (sum (phs%m_out) < phs%m_in(1)) then
           s_hat = phs%m_in(1) ** 2
           phs%f(1) = 1 / s_hat &
                 * sqrt (lambda (s_hat, phs%m_out(1)**2, phs%m_out(2)**2))
        else
           print *, "m_in  = ", phs%m_in
           print *, "m_out = ", phs%m_out
           call msg_fatal ("Decay is kinematically forbidden")
        end if
     end if
  case (2)
     if (phs%config%sqrts_fixed) then
        if (phs%p_defined)  return
        s_hat = phs%config%sqrts ** 2
     else
        if (.not. phs%p_defined)  return
        s_hat = sum (phs%p) ** 2
     end if
     if (sum (phs%m_in)**2 < s_hat .and. sum (phs%m_out)**2 < s_hat) then
        phs%f(1) = 1 / s_hat * &
              ( lambda (s_hat, phs%m_in (1)**2, phs%m_in (2)**2)   &
              * lambda (s_hat, phs%m_out(1)**2, phs%m_out(2)**2) ) &
              ** 0.25_default
     else
        phs%f(1) = 0
     end if
  end select
end subroutine phs_single_compute_factor
```

Evaluation. We uniformly generate $\cos\theta$ and ϕ, with Jacobian normalized to one.

There is only a single channel, so the second subroutine does nothing.

Note: the current implementation works for elastic scattering only.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: evaluate_selected_channel => phs_single_evaluate_selected_channel
procedure :: evaluate_other_channels => phs_single_evaluate_other_channels
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_evaluate_selected_channel (phs, c_in, r_in)
  class(phs_single_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  !!! Catching a gfortran bogus warning
  type(vector4_t), dimension(2) :: p_dum
  if (phs%p_defined) then
     call phs%select_channel (c_in)
     phs%r(:,c_in) = r_in
```

```
                  select case (phs%config%n_in)
                  case (2)
                     if (all (phs%m_in == phs%m_out)) then
                        call compute_kinematics_solid_angle (phs%p, phs%q, r_in)
                     else
                        call msg_bug ("PHS single: inelastic scattering not implemented")
                     end if
                  case (1)
                     !!! Catching a gfortran bogus warning
                     !!! call compute_kinematics_solid_angle (phs%decay_p (), phs%q, x)
                     p_dum = phs%decay_p ()
                     call compute_kinematics_solid_angle (p_dum, phs%q, r_in)
                  end select
                  call phs%compute_factor ()
                  phs%q_defined = .true.
                  phs%r_defined = .true.
               end if
            end subroutine phs_single_evaluate_selected_channel

            subroutine phs_single_evaluate_other_channels (phs, c_in)
              class(phs_single_t), intent(inout) :: phs
              integer, intent(in) :: c_in
            end subroutine phs_single_evaluate_other_channels
```

Auxiliary: split a decaying particle at rest into the decay products, aligned along
the z axis.
⟨*PHS single: phs single: TBP*⟩+≡
```
            procedure :: decay_p => phs_single_decay_p
```
⟨*PHS single: procedures*⟩+≡
```
            function phs_single_decay_p (phs) result (p)
              class(phs_single_t), intent(in) :: phs
              type(vector4_t), dimension(2) :: p
              real(default) :: k
              real(default), dimension(2) :: E
              k = sqrt (lambda (phs%m_in(1) ** 2, phs%m_out(1) ** 2, phs%m_out(2) ** 2)) &
                   / (2 * phs%m_in(1))
              E = sqrt (phs%m_out ** 2 + k ** 2)
              p(1) = vector4_moving (E(1), k, 3)
              p(2) = vector4_moving (E(2),-k, 3)
            end function phs_single_decay_p
```

Inverse evaluation.
⟨*PHS single: phs single: TBP*⟩+≡
```
            procedure :: inverse => phs_single_inverse
```
⟨*PHS single: procedures*⟩+≡
```
            subroutine phs_single_inverse (phs)
              class(phs_single_t), intent(inout) :: phs
              real(default), dimension(:), allocatable :: x
              if (phs%p_defined .and. phs%q_defined) then
                 call phs%select_channel ()
                 allocate (x (phs%config%n_par))
                 call inverse_kinematics_solid_angle (phs%p, phs%q, x)
```

```
        phs%r(:,1) = x
        call phs%compute_factor ()
        phs%r_defined = .true.
     end if
  end subroutine phs_single_inverse
```

11.3.3   Unit tests

⟨*PHS single: public*⟩+≡
```
    public :: phs_single_test
```

⟨*PHS single: tests*⟩≡
```
    subroutine phs_single_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*PHS single: execute tests*⟩
```
    end subroutine phs_single_test
```

Phase-space configuration data

Construct and display a test phase-space configuration object. Also check the
`azimuthal_dependence` flag.

⟨*PHS single: execute tests*⟩≡
```
    call test (phs_single_1, "phs_single_1", &
         "phase-space configuration", &
         u, results)
```

⟨*PHS single: tests*⟩+≡
```
    subroutine phs_single_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_constants_t) :: process_data
      class(phs_config_t), allocatable :: phs_data
      real(default) :: sqrts

      write (u, "(A)")  "* Test output: phs_single_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
           &phase-space configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)

      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_single_1"), process_data)
```

1398

```
    allocate (phs_single_config_t :: phs_data)
    call phs_data%init (process_data, model)

    sqrts = 1000._default
    call phs_data%configure (sqrts, azimuthal_dependence=.false.)

    call phs_data%write (u)

    call phs_data%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_single_1"

  end subroutine phs_single_1
```

Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

⟨*PHS single: execute tests*⟩+≡

```
    call test (phs_single_2, "phs_single_2", &
        "phase-space evaluation", &
        u, results)
```

⟨*PHS single: tests*⟩+≡

```
    subroutine phs_single_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(process_constants_t) :: process_data
      real(default) :: sqrts, E
      class(phs_config_t), allocatable, target :: phs_data
      class(phs_t), pointer :: phs => null ()
      type(vector4_t), dimension(2) :: p, q

      write (u, "(A)")  "* Test output: phs_single_2"
      write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
          var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)

      write (u, "(A)")  "* Initialize a process and a matching &
          &phase-space configuration"
      write (u, "(A)")
```

1399

```
call init_test_process_data (var_str ("phs_single_2"), process_data)

allocate (phs_single_config_t :: phs_data)
call phs_data%init (process_data, model)

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
    &for x = 0.5, 0.125"
write (u, "(A)")

call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
call phs%evaluate_other_channels (1)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
```

```
    call phs%set_outgoing_momenta (q)

    call phs%inverse ()
    call phs%write (u)

    call phs%final ()
    deallocate (phs)

    call phs_data%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_single_2"


  end subroutine phs_single_2
```

Phase space for non-c.m. system

Compute kinematics for given parameters, also invert the calculation. Since this will involve cancellations, we call `pacify` to eliminate numerical noise.

⟨*PHS single: execute tests*⟩+≡
```
    call test (phs_single_3, "phs_single_3", &
         "phase-space evaluation in lab frame", &
         u, results)
```

⟨*PHS single: tests*⟩+≡
```
    subroutine phs_single_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(process_constants_t) :: process_data
      real(default) :: sqrts, E
      class(phs_config_t), allocatable, target :: phs_data
      class(phs_t), pointer :: phs => null ()
      type(vector4_t), dimension(2) :: p, q
      type(lorentz_transformation_t) :: lt

      write (u, "(A)")  "* Test output: phs_single_3"
      write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
      write (u, "(A)")  "*            without c.m. kinematics assumption"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)

      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
```

1401

```
write (u, "(A)")

call init_test_process_data (var_str ("phs_single_3"), process_data)

allocate (phs_single_config_t :: phs_data)
call phs_data%init (process_data, model)

sqrts = 1000._default
call phs_data%configure (sqrts, cm_frame=.false., sqrts_fixed=.false.)

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta in lab system"
write (u, "(A)")

lt = boost (0.1_default, 1) * boost (0.3_default, 3)

E = sqrts / 2
p(1) = lt * vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = lt * vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call vector4_write (p(1), u)
call vector4_write (p(2), u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.5, 0.125"
write (u, "(A)")

call phs%set_incoming_momenta (p)
call phs%compute_flux ()

call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
call phs%evaluate_other_channels (1)
call pacify (phs)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract outgoing momenta in lab system"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
call vector4_write (q(1), u)
call vector4_write (q(2), u)
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Inverse kinematics"
      write (u, "(A)")

      deallocate (phs)
      call phs_data%allocate_instance (phs)
      call phs%init (phs_data)

      sqrts = 1000._default
      call phs_data%configure (sqrts)

      call phs%set_incoming_momenta (p)
      call phs%compute_flux ()
      call phs%set_outgoing_momenta (q)

      call phs%inverse ()
      call pacify (phs)
      call phs%write (u)

      call phs%final ()
      deallocate (phs)

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: phs_single_3"

    end subroutine phs_single_3
```

Decay Phase space evaluation

Compute kinematics for given parameters, also invert the calculation. This time, implement a decay process.

⟨*PHS single: execute tests*⟩+≡
```
    call test (phs_single_4, "phs_single_4", &
        "decay phase-space evaluation", &
        u, results)
```

⟨*PHS single: tests*⟩+≡
```
    subroutine phs_single_4 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(var_list_t), pointer :: var_list
      type(flavor_t) :: flv
      type(process_constants_t) :: process_data
      class(phs_config_t), allocatable, target :: phs_data
      class(phs_t), pointer :: phs => null ()
      type(vector4_t), dimension(1) :: p
```

```
type(vector4_t), dimension(2) :: q

write (u, "(A)")  "* Test output: phs_single_4"
write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
var_list => model_get_var_list_ptr (model)
call var_list_set_real (var_list, &
     var_str ("ff"), 0.4_default, is_known = .true.)
call model_parameters_update (model)
call flavor_init (flv, 25, model)

write (u, "(A)")  "* Initialize a decay and a matching &
     &phase-space configuration"
write (u, "(A)")

call init_test_decay_data (var_str ("phs_single_4"), process_data)

allocate (phs_single_config_t :: phs_data)
call phs_data%init (process_data, model)

call phs_data%configure (flavor_get_mass (flv))

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

p(1) = vector4_at_rest (flavor_get_mass (flv))

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.5, 0.125"
write (u, "(A)")

call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
```

```
      call phs%evaluate_other_channels (1)
      call phs%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Inverse kinematics"
      write (u, "(A)")

      call phs%get_outgoing_momenta (q)
      deallocate (phs)
      call phs_data%allocate_instance (phs)
      call phs%init (phs_data)

      call phs_data%configure (flavor_get_mass (flv))

      call phs%set_incoming_momenta (p)
      call phs%compute_flux ()
      call phs%set_outgoing_momenta (q)

      call phs%inverse ()
      call phs%write (u)

      call phs%final ()
      deallocate (phs)

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_single_4"

    end subroutine phs_single_4
```

11.4 Mappings

Mappings are objects that encode the transformation of the interval $(0, 1)$ to a physical variable m^2 or $\cos\theta$ (and back), as it is used in the phase space parameterization. The mapping objects contain fixed parameters, the associated methods implement the mapping and inverse mapping operations, including the computation of the Jacobian (phase space factor).

⟨`mappings.f90`⟩≡
  ⟨*File header*⟩

  ```
  module mappings
  ```

  ⟨*Use kinds*⟩
  ```
    use kinds, only: TC !NODEP!
  ```
  ⟨*Use strings*⟩
  ```
    use limits, only: FMT_19 !NODEP!
    use constants, only: pi !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use md5
    use models
    use flavors
  ```

  ⟨*Standard module head*⟩

  ⟨*Mappings: public*⟩

  ⟨*Mappings: parameters*⟩

  ⟨*Mappings: types*⟩

  ⟨*Mappings: interfaces*⟩

  ```
  contains
  ```

  ⟨*Mappings: procedures*⟩

  ```
  end module mappings
  ```

11.4.1 Default parameters

This type holds the default parameters, needed for setting the scale in cases where no mass parameter is available. The contents are public.

⟨*Mappings: public*⟩≡
  ```
      public :: mapping_defaults_t
  ```

⟨*Mappings: types*⟩≡
  ```
      type :: mapping_defaults_t
         real(default) :: energy_scale = 10
         real(default) :: invariant_mass_scale = 10
         real(default) :: momentum_transfer_scale = 10
         logical :: step_mapping = .true.
         logical :: step_mapping_exp = .true.
  ```

```
      logical :: enable_s_mapping = .false.
    contains
    ⟨Mappings: mapping defaults: TBP⟩
    end type mapping_defaults_t
```

Output.

⟨*Mappings: mapping defaults: TBP*⟩≡
```
    procedure :: write => mapping_defaults_write
```

⟨*Mappings: procedures*⟩≡
```
    subroutine mapping_defaults_write (object, unit)
      class(mapping_defaults_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A," // FMT_19 // ")") "energy scale  = ", &
           object%energy_scale
      write (u, "(3x,A," // FMT_19 // ")") "mass scale    = ", &
           object%invariant_mass_scale
      write (u, "(3x,A," // FMT_19 // ")") "q scale       = ", &
           object%momentum_transfer_scale
      write (u, "(3x,A,L1)") "step mapping    = ", &
           object%step_mapping
      write (u, "(3x,A,L1)") "step exp. mode  = ", &
           object%step_mapping_exp
      write (u, "(3x,A,L1)") "allow s mapping = ", &
           object%enable_s_mapping
    end subroutine mapping_defaults_write
```

⟨*Mappings: public*⟩+≡
```
    public :: mapping_defaults_md5sum
```

⟨*Mappings: procedures*⟩+≡
```
    function mapping_defaults_md5sum (mapping_defaults) result (md5sum_map)
      character(32) :: md5sum_map
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      integer :: u
      u = free_unit ()
      open (u, status = "scratch")
      write (u, *)  mapping_defaults%energy_scale
      write (u, *)  mapping_defaults%invariant_mass_scale
      write (u, *)  mapping_defaults%momentum_transfer_scale
      write (u, *)  mapping_defaults%step_mapping
      write (u, *)  mapping_defaults%step_mapping_exp
      write (u, *)  mapping_defaults%enable_s_mapping
      rewind (u)
      md5sum_map = md5sum (u)
      close (u)
    end function mapping_defaults_md5sum
```

11.4.2 The Mapping type

Each mapping has a type (e.g., s-channel, infrared), a binary code (redundant, but useful for debugging), and a reference particle. The flavor code of this particle is stored for bookkeeping reasons, what matters are the mass and width of this particle. Furthermore, depending on the type, various mapping parameters can be set and used.

The parameters `a1` to `a3` (for m^2 mappings) and `b1` to `b3` (for $\cos\theta$ mappings) are values that are stored once to speed up the calculation, if `variable_limits` is false. The exact meaning of these parameters depends on the mapping type. The limits are fixed if there is a fixed c.m. energy.

⟨*Mappings: public*⟩+≡
```
    public :: mapping_t
```

⟨*Mappings: types*⟩+≡
```
    type :: mapping_t
      private
      integer :: type = NO_MAPPING
      integer(TC) :: bincode
      type(flavor_t) :: flv
      real(default) :: mass = 0
      real(default) :: width = 0
      logical :: a_unknown = .true.
      real(default) :: a1 = 0
      real(default) :: a2 = 0
      real(default) :: a3 = 0
      logical :: b_unknown = .true.
      real(default) :: b1 = 0
      real(default) :: b2 = 0
      real(default) :: b3 = 0
      logical :: variable_limits = .true.
    end type mapping_t
```

The valid mapping types. The extra type `STEP_MAPPING` is used only internally.

⟨*Mappings: parameters*⟩≡
  ⟨*Mapping modes*⟩

11.4.3 Screen output

Do not write empty mappings.

⟨*Mappings: public*⟩+≡
```
    public :: mapping_write
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_write (map, unit, verbose)
      type(mapping_t), intent(in) :: map
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      character(len=9) :: str
      u = output_unit (unit);  if (u < 0)  return
      select case(map%type)
      case(S_CHANNEL); str = "s_channel"
```

```
            case(COLLINEAR); str = "collinear"
            case(INFRARED);  str = "infrared "
            case(RADIATION); str = "radiation"
            case(T_CHANNEL); str = "t_channel"
            case(U_CHANNEL); str = "u_channel"
            case(STEP_MAPPING_E);  str = "step_exp"
            case(STEP_MAPPING_H);  str = "step_hyp"
            case(ON_SHELL);  str = "on_shell"
            case default;    str = "????????"
         end select
         if (map%type /= NO_MAPPING) then
            write (u, '(1x,A,I4,A)') &
                  "Branch #", map%bincode, ":  " // &
                  "Mapping (" // str // ") for particle " // &
                  '"' // char (flavor_get_name (map%flv)) // '"'
            if (present (verbose)) then
               if (verbose) then
                  select case (map%type)
                  case (S_CHANNEL, RADIATION, STEP_MAPPING_E, STEP_MAPPING_H)
                     write (u, "(1x,A,3(" // FMT_19 // "))")  &
                           " m/w    = ", map%mass, map%width
                  case default
                     write (u, "(1x,A,3(" // FMT_19 // "))")  &
                           " m      = ", map%mass
                  end select
                  select case (map%type)
                  case (S_CHANNEL, T_CHANNEL, U_CHANNEL, &
                        STEP_MAPPING_E, STEP_MAPPING_H, &
                        COLLINEAR, INFRARED, RADIATION)
                     write (u, "(1x,A,3(" // FMT_19 // "))")  &
                           " a1/2/3 = ", map%a1, map%a2, map%a3
                  end select
                  select case (map%type)
                  case (T_CHANNEL, U_CHANNEL, COLLINEAR)
                     write (u, "(1x,A,3(" // FMT_19 // "))")  &
                           " b1/2/3 = ", map%b1, map%b2, map%b3
                  end select
               end if
            end if
         end if
      end subroutine mapping_write
```

11.4.4 Define a mapping

The initialization routine sets the mapping type and the particle (binary code and flavor code) for which the mapping applies (e.g., a Z resonance in branch #3). We only need the absolute value of the flavor code.

⟨*Mappings: public*⟩+≡
```
    public :: mapping_init
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_init (mapping, bincode, type, f, model)
      type(mapping_t), intent(inout) :: mapping
```

```
    integer(TC), intent(in) :: bincode
    type(string_t), intent(in) :: type
    integer, intent(in), optional :: f
    type(model_t), intent(in), optional, target :: model
    mapping%bincode = bincode
    select case (char (type))
    case ("s_channel");  mapping%type = S_CHANNEL
    case ("collinear");  mapping%type = COLLINEAR
    case ("infrared");   mapping%type = INFRARED
    case ("radiation");  mapping%type = RADIATION
    case ("t_channel");  mapping%type = T_CHANNEL
    case ("u_channel");  mapping%type = U_CHANNEL
    case ("step_exp");   mapping%type = STEP_MAPPING_E
    case ("step_hyp");   mapping%type = STEP_MAPPING_H
    case ("on_shell");   mapping%type = ON_SHELL
    case default
       call msg_bug ("Mappings: encountered undefined mapping key '" &
             // char (type) // "'")
    end select
    if (present (f) .and. present (model)) &
         call flavor_init (mapping%flv, abs (f), model)
 end subroutine mapping_init
```

This sets the actual mass and width, using a parameter set. Since the auxiliary parameters will only be determined when the mapping is first called, they are marked as unknown.

⟨*Mappings: public*⟩+≡

```
    public :: mapping_set_parameters
```

⟨*Mappings: procedures*⟩+≡

```
    subroutine mapping_set_parameters (map, mapping_defaults, variable_limits)
      type(mapping_t), intent(inout) :: map
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      logical, intent(in) :: variable_limits
      if (map%type /= NO_MAPPING) then
         map%mass  = flavor_get_mass (map%flv)
         map%width = flavor_get_width (map%flv)
         map%variable_limits = variable_limits
         map%a_unknown = .true.
         map%b_unknown = .true.
         select case (map%type)
         case (S_CHANNEL)
            if (map%mass <= 0) then
               call mapping_write (map)
               call msg_fatal &
                    & (" S-channel resonance must have positive mass")
            else if (map%width <= 0) then
               call mapping_write (map)
               call msg_fatal &
                    & (" S-channel resonance must have positive width")
            end if
         case (RADIATION)
            map%width = max (map%width, mapping_defaults%energy_scale)
         case (INFRARED, COLLINEAR)
```

```
          map%mass = max (map%mass, mapping_defaults%invariant_mass_scale)
       case (T_CHANNEL, U_CHANNEL)
          map%mass = max (map%mass, mapping_defaults%momentum_transfer_scale)
       end select
    end if
  end subroutine mapping_set_parameters
```

For a step mapping the mass and width are set directly, instead of being deter-
mined from the flavor parameter (which is meaningless here). They correspond
to the effective upper bound of phase space due to a resonance, as opposed to
the absolute upper bound.

⟨*Mappings: public*⟩+≡

```
  public :: mapping_set_step_mapping_parameters
```

⟨*Mappings: procedures*⟩+≡

```
  subroutine mapping_set_step_mapping_parameters (map, &
      mass, width, variable_limits)
    type(mapping_t), intent(inout) :: map
    real(default), intent(in) :: mass, width
    logical, intent(in) :: variable_limits
    select case (map%type)
    case (STEP_MAPPING_E, STEP_MAPPING_H)
       map%variable_limits = variable_limits
       map%a_unknown = .true.
       map%b_unknown = .true.
       map%mass = mass
       map%width = width
    end select
  end subroutine mapping_set_step_mapping_parameters
```

11.4.5   Retrieve contents

Return true if there is any / an s-channel mapping.

⟨*Mappings: public*⟩+≡

```
  public :: mapping_is_set
  public :: mapping_is_s_channel
  public :: mapping_is_on_shell
```

⟨*Mappings: procedures*⟩+≡

```
  function mapping_is_set (mapping) result (flag)
    logical :: flag
    type(mapping_t), intent(in) :: mapping
    flag = mapping%type /= NO_MAPPING
  end function mapping_is_set

  function mapping_is_s_channel (mapping) result (flag)
    logical :: flag
    type(mapping_t), intent(in) :: mapping
    flag = mapping%type == S_CHANNEL
  end function mapping_is_s_channel

  function mapping_is_on_shell (mapping) result (flag)
```

```
    logical :: flag
    type(mapping_t), intent(in) :: mapping
    flag = mapping%type == ON_SHELL
  end function mapping_is_on_shell
```

Return stored mass and width, respectively.

⟨*Mappings: public*⟩+≡
```
    public :: mapping_get_mass
    public :: mapping_get_width
```

⟨*Mappings: procedures*⟩+≡
```
    function mapping_get_mass (mapping) result (mass)
      real(default) :: mass
      type(mapping_t), intent(in) :: mapping
      mass = mapping%mass
    end function mapping_get_mass

    function mapping_get_width (mapping) result (width)
      real(default) :: width
      type(mapping_t), intent(in) :: mapping
      width = mapping%width
    end function mapping_get_width
```

11.4.6 Compare mappings

Equality for single mappings and arrays

⟨*Mappings: public*⟩+≡
```
    public :: operator(==)
```

⟨*Mappings: interfaces*⟩≡
```
    interface operator(==)
       module procedure mapping_equal
    end interface
```

⟨*Mappings: procedures*⟩+≡
```
    function mapping_equal (m1, m2) result (equal)
      type(mapping_t), intent(in) :: m1, m2
      logical :: equal
      if (m1%type == m2%type) then
         select case (m1%type)
         case (NO_MAPPING)
            equal = .true.
         case (S_CHANNEL, RADIATION, STEP_MAPPING_E, STEP_MAPPING_H)
            equal = (m1%mass == m2%mass) .and. (m1%width == m2%width)
         case default
            equal = (m1%mass == m2%mass)
         end select
      else
         equal = .false.
      end if
    end function mapping_equal
```

11.4.7 Mappings of the invariant mass

Inserting an x value between 0 and 1, we want to compute the corresponding invariant mass $m^2(x)$ and the jacobian, aka phase space factor $f(x)$. We also need the reverse operation.

In general, the phase space factor f is defined by

$$\frac{1}{s} \int_{m_{\min}^2}^{m_{\max}^2} dm^2\, g(m^2) = \int_0^1 dx\, \frac{1}{s}\frac{dm^2}{dx}\, g(m^2(x)) = \int_0^1 dx\, f(x)\, g(x), \qquad (11.8)$$

where thus

$$f(x) = \frac{1}{s}\frac{dm^2}{dx}. \qquad (11.9)$$

With this mapping, a function of the form

$$g(m^2) = c\frac{dx(m^2)}{dm^2} \qquad (11.10)$$

is mapped to a constant:

$$\frac{1}{s} \int_{m_{\min}^2}^{m_{\max}^2} dm^2\, g(m^2) = \int_0^1 dx\, f(x)\, g(m^2(x)) = \int_0^1 dx\, \frac{c}{s}. \qquad (11.11)$$

Here is the mapping routine. Input are the available energy squared `s`, the limits for m^2, and the x value. Output are the m^2 value and the phase space factor f.

⟨*Mappings: public*⟩+≡
```
    public :: mapping_compute_msq_from_x
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_compute_msq_from_x (map, s, msq_min, msq_max, msq, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s, msq_min, msq_max
      real(default), intent(out) :: msq, f
      real(default), intent(in) :: x
      real(default) :: z, msq0, msq1, tmp
      integer :: type
      type = map%type
      if (s == 0) &
          call msg_fatal (" Applying msq mapping for zero energy")
```
⟨*Modify mapping type if necessary*⟩
```
      select case(type)
      case (NO_MAPPING)
```
   ⟨*Constants for trivial msq mapping*⟩
   ⟨*Apply trivial msq mapping*⟩
```
      case (S_CHANNEL)
```
   ⟨*Constants for s-channel resonance mapping*⟩
   ⟨*Apply s-channel resonance mapping*⟩
```
      case (COLLINEAR, INFRARED, RADIATION)
```
   ⟨*Constants for s-channel pole mapping*⟩
   ⟨*Apply s-channel pole mapping*⟩
```
      case (T_CHANNEL, U_CHANNEL)
```
   ⟨*Constants for t-channel pole mapping*⟩
   ⟨*Apply t-channel pole mapping*⟩

1413

```
    case (STEP_MAPPING_E)
     ⟨Constants for exponential step mapping⟩
     ⟨Apply exponential step mapping⟩
    case (STEP_MAPPING_H)
     ⟨Constants for hyperbolic step mapping⟩
     ⟨Apply hyperbolic step mapping⟩
    case default
       call msg_fatal ( " Attempt to apply undefined msq mapping")
    end select
end subroutine mapping_compute_msq_from_x
```

The inverse mapping

⟨*Mappings: public*⟩+≡
```
    public :: mapping_compute_x_from_msq
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_compute_x_from_msq (map, s, msq_min, msq_max, msq, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s, msq_min, msq_max
      real(default), intent(in) :: msq
      real(default), intent(out) :: f, x
      real(default) :: msq0, msq1, tmp, z
      integer :: type
      type = map%type
      if (s == 0) &
           call msg_fatal (" Applying inverse msq mapping for zero energy")
```
⟨*Modify mapping type if necessary*⟩
```
      select case (type)
      case (NO_MAPPING)
       ⟨Constants for trivial msq mapping⟩
       ⟨Apply inverse trivial msq mapping⟩
      case (S_CHANNEL)
       ⟨Constants for s-channel resonance mapping⟩
       ⟨Apply inverse s-channel resonance mapping⟩
      case (COLLINEAR, INFRARED, RADIATION)
       ⟨Constants for s-channel pole mapping⟩
       ⟨Apply inverse s-channel pole mapping⟩
      case (T_CHANNEL, U_CHANNEL)
       ⟨Constants for t-channel pole mapping⟩
       ⟨Apply inverse t-channel pole mapping⟩
      case (STEP_MAPPING_E)
       ⟨Constants for exponential step mapping⟩
       ⟨Apply inverse exponential step mapping⟩
      case (STEP_MAPPING_H)
       ⟨Constants for hyperbolic step mapping⟩
       ⟨Apply inverse hyperbolic step mapping⟩
      case default
         call msg_fatal ( " Attempt to apply undefined msq mapping")
      end select
    end subroutine mapping_compute_x_from_msq
```

Trivial mapping

We simply map the boundaries of the interval $(m_{\min}, m_{\max})$ to $(0, 1)$:

$$m^2 = (1-x)m_{\min}^2 + xm_{\max}^2; \qquad (11.12)$$

the inverse is

$$x = \frac{m^2 - m_{\min}^2}{m_{\max}^2 - m_{\min}^2}. \qquad (11.13)$$

Hence

$$f(x) = \frac{m_{\max}^2 - m_{\min}^2}{s}, \qquad (11.14)$$

and we have, as required,

$$f(x)\frac{dx}{dm^2} = \frac{1}{s}. \qquad (11.15)$$

We store the constant parameters the first time the mapping is called – or, if limits vary, recompute them each time.

⟨*Constants for trivial msq mapping*⟩≡
```
if (map%variable_limits .or. map%a_unknown) then
   map%a1 = 0
   map%a2 = msq_max - msq_min
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply trivial msq mapping*⟩≡
```
msq = (1-x) * msq_min + x * msq_max
f = map%a3
```

⟨*Apply inverse trivial msq mapping*⟩≡
```
if (map%a2 /= 0) then
   x = (msq - msq_min) / map%a2
else
   x = 0
end if
f = map%a3
```

Resonance or step mapping does not make much sense if the resonance mass is outside the kinematical bounds. If this is the case, revert to `NO_MAPPING`. This is possible even if the kinematical bounds vary from event to event.

⟨*Modify mapping type if necessary*⟩≡
```
select case (type)
case (S_CHANNEL, STEP_MAPPING_E, STEP_MAPPING_H)
   msq0 = map%mass**2
   if (msq0 < msq_min .or. msq0 > msq_max)  type = NO_MAPPING
end select
```

Breit-Wigner mapping

A Breit-Wigner resonance with mass M and width Γ is flattened by the following mapping:

This mapping does not make much sense if the resonance mass is too low. If this is the case, revert to `NO_MAPPING`. There is a tricky point with this if the

1415

mass is too high: `msq_max` is not a constant if structure functions are around. However, switching the type depending on the overall energy does not change the integral, it is just another branching point.

$$m^2 = M(M + t\Gamma), \qquad (11.16)$$

where

$$t = \tan\left[(1-x)\arctan\frac{m_{\min}^2 - M^2}{M\Gamma} + x\arctan\frac{m_{\max}^2 - M^2}{M\Gamma}\right]. \qquad (11.17)$$

The inverse:

$$x = \frac{\arctan\frac{m^2 - M^2}{M\Gamma} - \arctan\frac{m_{\min}^2 - M^2}{M\Gamma}}{\arctan\frac{m_{\max}^2 - M^2}{M\Gamma} - \arctan\frac{m_{\min}^2 - M^2}{M\Gamma}} \qquad (11.18)$$

The phase-space factor of this transformation is

$$f(x) = \frac{M\Gamma}{s}\left(\arctan\frac{m_{\max}^2 - M^2}{M\Gamma} - \arctan\frac{m_{\min}^2 - M^2}{M\Gamma}\right)(1 + t^2). \qquad (11.19)$$

This maps any function proportional to

$$g(m^2) = \frac{M\Gamma}{(m^2 - M^2)^2 + M^2\Gamma^2} \qquad (11.20)$$

to a constant times $1/s$.

⟨*Constants for s-channel resonance mapping*⟩≡
```
if (map%variable_limits .or. map%a_unknown) then
   msq0 = map%mass ** 2
   map%a1 = atan ((msq_min - msq0) / (map%mass * map%width))
   map%a2 = atan ((msq_max - msq0) / (map%mass * map%width))
   map%a3 = (map%a2 - map%a1) * (map%mass * map%width) / s
   map%a_unknown = .false.
end if
```

⟨*Apply s-channel resonance mapping*⟩≡
```
z = (1-x) * map%a1 + x * map%a2
if (-pi/2 < z .and. z < pi/2) then
   tmp = tan (z)
   msq = map%mass * (map%mass + map%width * tmp)
   f = map%a3 * (1 + tmp**2)
else
   msq = 0
   f = 0
end if
```

⟨*Apply inverse s-channel resonance mapping*⟩≡
```
tmp = (msq - msq0) / (map%mass * map%width)
x = (atan (tmp) - map%a1) / (map%a2 - map%a1)
f = map%a3 * (1 + tmp**2)
```

Mapping for massless splittings

This mapping accounts for approximately scale-invariant behavior where $\ln M^2$ is evenly distributed.

$$m^2 = m_{\min}^2 + M^2 \left(\exp(xL) - 1 \right) \tag{11.21}$$

where

$$L = \ln \left(\frac{m_{\max}^2 - m_{\min}^2}{M^2} + 1 \right). \tag{11.22}$$

The inverse:

$$x = \frac{1}{L} \ln \left(\frac{m^2 - m_{\min}^2}{M^2} + 1 \right) \tag{11.23}$$

The constant M is a characteristic scale. Above this scale $(m^2 - m_{\min}^2 \gg M^2)$, this mapping behaves like $x \propto \ln m^2$, while below the scale it reverts to a linear mapping.

The phase-space factor is

$$f(x) = \frac{M^2}{s} \exp(xL)\, L. \tag{11.24}$$

A function proportional to

$$g(m^2) = \frac{1}{(m^2 - m_{\min}^2) + M^2} \tag{11.25}$$

is mapped to a constant, i.e., a simple pole near $m_{\min}$ with a regulator mass M.

This type of mapping is useful for massless collinear and infrared singularities, where the scale is stored as the mass parameter. In the radiation case (IR radiation off massive particle), the heavy particle width is the characteristic scale.

⟨*Constants for s-channel pole mapping*⟩≡
```
if (map%variable_limits .or. map%a_unknown) then
   if (type == RADIATION) then
      msq0 = map%width**2
   else
      msq0 = map%mass**2
   end if
   map%a1 = msq0
   map%a2 = log ((msq_max - msq_min) / msq0 + 1)
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply s-channel pole mapping*⟩≡
```
msq1 = map%a1 * exp (x * map%a2)
msq = msq1 - map%a1 + msq_min
f = map%a3 * msq1
```

⟨*Apply inverse s-channel pole mapping*⟩≡
```
msq1 = msq - msq_min + map%a1
x = log (msq1 / map%a1) / map%a2
f = map%a3 * msq1
```

Mapping for t-channel poles

This is also approximately scale-invariant, and we use the same type of mapping as before. However, we map $1/x$ singularities at both ends of the interval; again, the mapping becomes linear when the distance is less than M^2:

$$m^2 = \begin{cases} m_{\min}^2 + M^2 \left(\exp(xL) - 1 \right) & \text{for } 0 < x < \frac{1}{2} \\ m_{\max}^2 - M^2 \left(\exp((1-x)L) - 1 \right) & \text{for } \frac{1}{2} \le x < 1 \end{cases} \tag{11.26}$$

where

$$L = 2 \ln \left(\frac{m_{\max}^2 - m_{\min}^2}{2M^2} + 1 \right). \tag{11.27}$$

The inverse:

$$x = \begin{cases} \frac{1}{L} \ln \left(\frac{m^2 - m_{\min}^2}{M^2} + 1 \right) & \text{for } m^2 < (m_{\max}^2 - m_{\min}^2)/2 \\ 1 - \frac{1}{L} \ln \left(\frac{m_{\max} - m^2}{M^2} + 1 \right) & \text{for } m^2 \ge (m_{\max}^2 - m_{\min}^2)/2 \end{cases} \tag{11.28}$$

The phase-space factor is

$$f(x) = \begin{cases} \frac{M^2}{s} \exp(xL) \, L. & \text{for } 0 < x < \frac{1}{2} \\ \frac{M^2}{s} \exp((1-x)L) \, L. & \text{for } \frac{1}{2} \le x < 1 \end{cases} \tag{11.29}$$

A (continuous) function proportional to

$$g(m^2) = \begin{cases} 1/(m^2 - m_{\min}^2) + M^2) & \text{for } m^2 < (m_{\max}^2 - m_{\min}^2)/2 \\ 1/((m_{\max}^2 - m^2) + M^2) & \text{for } m^2 \le (m_{\max}^2 - m_{\min}^2)/2 \end{cases} \tag{11.30}$$

is mapped to a constant by this mapping, i.e., poles near both ends of the interval.

⟨*Constants for t-channel pole mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   msq0 = map%mass**2
   map%a1 = msq0
   map%a2 = 2 * log ((msq_max - msq_min)/(2*msq0) + 1)
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply t-channel pole mapping*⟩≡

```
if (x < .5_default) then
   msq1 = map%a1 * exp (x * map%a2)
   msq = msq1 - map%a1 + msq_min
else
   msq1 = map%a1 * exp ((1-x) * map%a2)
   msq = -(msq1 - map%a1) + msq_max
end if
f = map%a3 * msq1
```

⟨*Apply inverse t-channel pole mapping*⟩≡

```
if (msq < (msq_max + msq_min)/2) then
   msq1 = msq - msq_min + map%a1
   x = log (msq1/map%a1) / map%a2
```

```
      else
         msq1 = msq_max - msq + map%a1
         x = 1 - log (msq1/map%a1) / map%a2
      end if
      f = map%a3 * msq1
```

11.4.8   Step mapping

Step mapping is useful when the allowed range for a squared-mass variable is large, but only a fraction at the lower end is populated because the particle in question is an (off-shell) decay product of a narrow resonance. I.e., if the resonance was forced to be on-shell, the upper end of the range would be the resonance mass, minus the effective (real or resonance) mass of the particle(s) in the sibling branch of the decay.

The edge of this phase space section has a width which is determined by the width of the parent, plus the width of the sibling branch. (The widths might be added in quadrature, but this precision is probably not important.)

Fermi function

A possible mapping is derived from the Fermi function which has precisely this behavior. The Fermi function is given by

$$f(x) = \frac{1}{1 + \exp \frac{x - \mu}{\gamma}} \tag{11.31}$$

where x is taken as the invariant mass squared, μ is the invariant mass squared of the edge, and γ is the effective width which is given by the widths of the parent and the sibling branch. (Widths might be added in quadrature, but we do not require this level of precision.)

$$x = \frac{m^2 - m_{\text{min}}^2}{\Delta m^2} \tag{11.32}$$

$$\mu = \frac{m_{\text{max,eff}}^2 - m_{\text{min}}^2}{\Delta m^2} \tag{11.33}$$

$$\gamma = \frac{2m_{\text{max,eff}}\Gamma}{\Delta m^2} \tag{11.34}$$

with

$$\Delta m^2 = m_{\text{max}}^2 - m_{\text{min}}^2 \tag{11.35}$$

m^2 is thus given by

$$m^2(x) = xm_{\text{max}}^2 + (1 - x)m_{\text{min}}^2 \tag{11.36}$$

For the mapping, we compute the integral $g(x)$ of the Fermi function, normalized such that $g(0) = 0$ and $g(1) = 1$. We introduce the abbreviations

$$\alpha = 1 - \gamma \ln \frac{1 + \beta e^{1/\gamma}}{1 + \beta} \tag{11.37}$$

$$\beta = e^{-\mu/\gamma} \tag{11.38}$$

and obtain

$$g(x) = \frac{1}{\alpha} \left(x - \gamma \ln \frac{1 + \beta e^{x/\gamma}}{1 + \beta} \right) \tag{11.39}$$

The actual mapping is the inverse function $h(y) = g^{-1}(y)$,

$$h(y) = -\gamma \ln \left(e^{-\alpha y/\gamma} (1 + \beta) - \beta \right) \tag{11.40}$$

The Jacobian is

$$\frac{dh}{dy} = \alpha \left(1 - e^{\alpha y/\gamma} \frac{\beta}{1 + \beta} \right)^{-1} \tag{11.41}$$

which is equal to $1/(dg/dx)$, namely

$$\frac{dg}{dx} = \frac{1}{\alpha} \frac{1}{1 + \beta e^{x/\gamma}} \tag{11.42}$$

The final result is

$$\int_{m_{\min}^2}^{m_{\max}^2} dm^2 \, F(m^2) = \Delta m^2 \int_0^1 dx \, F(m^2(x)) \tag{11.43}$$

$$= \Delta m^2 \int_0^1 dy \, F(m^2(h(y))) \frac{dh}{dy} \tag{11.44}$$

Here is the implementation. We fill a1, a2, a3 with α, β, γ, respectively.

⟨*Constants for exponential step mapping*⟩≡
```
    if (map%variable_limits .or. map%a_unknown) then
      map%a3 = max (2 * map%mass * map%width / (msq_max - msq_min), 0.01_default)
      map%a2 = exp (- (map%mass**2 - msq_min) / (msq_max - msq_min) &
                    / map%a3)
      map%a1 = 1 - map%a3 * log ((1 + map%a2 * exp (1 / map%a3)) / (1 + map%a2))
    end if
```

⟨*Apply exponential step mapping*⟩≡
```
    tmp = exp (- x * map%a1 / map%a3) * (1 + map%a2)
    z = - map%a3 * log (tmp - map%a2)
    msq  = z * msq_max + (1 - z) * msq_min
    f = map%a1 / (1 - map%a2 / tmp) * (msq_max - msq_min) / s
```

⟨*Apply inverse exponential step mapping*⟩≡
```
    z = (msq - msq_min) / (msq_max - msq_min)
    tmp = 1 + map%a2 * exp (z / map%a3)
    x = (z - map%a3 * log (tmp / (1 + map%a2))) &
        / map%a1
    f = map%a1 * tmp * (msq_max - msq_min) / s
```

Hyperbolic mapping

The Fermi function has the drawback that it decreases exponentially. It might be preferable to take a function with a power-law decrease, such that the high-mass region is not completely depopulated.

Here, we start with the actual mapping which we take as

$$h(y) = \frac{b}{a - y} - \frac{b}{a} + \mu y \tag{11.45}$$

with the abbreviation

$$a = \frac{1}{2}\left(1 + \sqrt{1 + \frac{4b}{1 - \mu}}\right) \tag{11.46}$$

This is a hyperbola in the xy plane. The derivative is

$$\frac{dh}{dy} = \frac{b}{(a-y)^2} + \mu \tag{11.47}$$

The constants correspond to

$$\mu = \frac{m^2_{\mathrm{max,eff}} - m^2_{\mathrm{min}}}{\Delta m^2} \tag{11.48}$$

$$b = \frac{1}{\mu}\left(\frac{2m_{\mathrm{max,eff}}\Gamma}{\Delta m^2}\right)^2 \tag{11.49}$$

The inverse function is the solution of a quadratic equation,

$$g(x) = \frac{1}{2}\left[\left(a + \frac{x}{\mu} + \frac{b}{a\mu}\right) - \sqrt{\left(a - \frac{x}{\mu}\right)^2 + 2\frac{b}{a\mu}\left(a + \frac{x}{\mu}\right) + \left(\frac{b}{a\mu}\right)^2}\right] \tag{11.50}$$

The constants $a_{1,2,3}$ are identified with a, b, μ.

⟨*Constants for hyperbolic step mapping*⟩≡
```
if (map%variable_limits .or. map%a_unknown) then
  map%a3 = (map%mass**2 - msq_min) / (msq_max - msq_min)
  map%a2 = max ((2 * map%mass * map%width / (msq_max - msq_min))**2 &
                / map%a3, 1e-6_default)
  map%a1 = (1 + sqrt (1 + 4 * map%a2 / (1 - map%a3))) / 2
end if
```

⟨*Apply hyperbolic step mapping*⟩≡
```
z = map%a2 / (map%a1 - x) - map%a2 / map%a1 + map%a3 * x
msq = z * msq_max + (1 - z) * msq_min
f = (map%a2 / (map%a1 - x)**2 + map%a3) * (msq_max - msq_min) / s
```

⟨*Apply inverse hyperbolic step mapping*⟩≡
```
z = (msq - msq_min) / (msq_max - msq_min)
tmp = map%a2 / (map%a1 * map%a3)
x = ((map%a1 + z / map%a3 + tmp) &
    - sqrt ((map%a1 - z / map%a3)**2 + 2 * tmp * (map%a1 + z / map%a3) &
            + tmp**2)) / 2
f = (map%a2 / (map%a1 - x)**2 + map%a3) * (msq_max - msq_min) / s
```

11.4.9   Mappings of the polar angle

The other type of singularity, a simple pole just outside the integration region, can occur in the integration over $\cos\theta$. This applies to exchange of massless (or light) particles.

Double poles (Coulomb scattering) are also possible, but only in certain cases. These are also handled by the single-pole mapping.

The mapping is analogous to the previous m^2 pole mapping, but with a different normalization and notation of variables:

$$\frac{1}{2} \int_{-1}^{1} d\cos\theta \, g(\theta) = \int_{0}^{1} dx \, \frac{d\cos\theta}{dx} \, g(\theta(x)) = \int_{0}^{1} dx \, f(x) \, g(x), \qquad (11.51)$$

where thus

$$f(x) = \frac{1}{2} \frac{d\cos\theta}{dx}. \qquad (11.52)$$

With this mapping, a function of the form

$$g(\theta) = c \frac{dx(\cos\theta)}{d\cos\theta} \qquad (11.53)$$

is mapped to a constant:

$$\int_{-1}^{1} d\cos\theta \, g(\theta) = \int_{0}^{1} dx \, f(x) \, g(\theta(x)) = \int_{0}^{1} dx \, c. \qquad (11.54)$$

⟨*Mappings: public*⟩+≡
```
    public :: mapping_compute_ct_from_x
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_compute_ct_from_x (map, s, ct, st, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s
      real(default), intent(out) :: ct, st, f
      real(default), intent(in) :: x
      real(default) :: tmp, ct1
      select case (map%type)
      case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION, &
            STEP_MAPPING_E, STEP_MAPPING_H)
```
      ⟨*Apply trivial ct mapping*⟩
```
      case (T_CHANNEL, U_CHANNEL, COLLINEAR)
```
      ⟨*Constants for ct pole mapping*⟩
      ⟨*Apply ct pole mapping*⟩
```
      case default
         call msg_fatal (" Attempt to apply undefined ct mapping")
      end select
    end subroutine mapping_compute_ct_from_x
```

⟨*Mappings: public*⟩+≡
```
    public :: mapping_compute_x_from_ct
```

⟨*Mappings: procedures*⟩+≡
```
    subroutine mapping_compute_x_from_ct (map, s, ct, f, x)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: s
      real(default), intent(in) :: ct
      real(default), intent(out) :: f, x
      real(default) :: ct1
      select case (map%type)
      case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION, &
            STEP_MAPPING_E, STEP_MAPPING_H)
```
      ⟨*Apply inverse trivial ct mapping*⟩

```
      case (T_CHANNEL, U_CHANNEL, COLLINEAR)
      ⟨Constants for ct pole mapping⟩
      ⟨Apply inverse ct pole mapping⟩
   case default
      call msg_fatal (" Attempt to apply undefined inverse ct mapping")
   end select
end subroutine mapping_compute_x_from_ct
```

Trivial mapping

This is just the mapping of the interval $(-1, 1)$ to $(0, 1)$:

$$\cos \theta = -1 + 2x \tag{11.55}$$

and

$$f(x) = 1 \tag{11.56}$$

with the inverse

$$x = \frac{1 + \cos \theta}{2} \tag{11.57}$$

⟨Apply trivial ct mapping⟩≡
```
tmp = 2 * (1-x)
ct = 1 - tmp
st = sqrt (tmp * (2-tmp))
f = 1
```

⟨Apply inverse trivial ct mapping⟩≡
```
x = (ct + 1) / 2
f = 1
```

Pole mapping

As above for m^2, we simultaneously map poles at both ends of the $\cos \theta$ interval. The formulae are completely analogous:

$$\cos \theta = \begin{cases} \frac{M^2}{s} \left[\exp(xL) - 1 \right] - 1 & \text{for } x < \frac{1}{2} \\ -\frac{M^2}{s} \left[\exp((1-x)L) - 1 \right] + 1 & \text{for } x \geq \frac{1}{2} \end{cases} \tag{11.58}$$

where

$$L = 2 \ln \frac{M^2 + s}{M^2}. \tag{11.59}$$

Inverse:

$$x = \begin{cases} \frac{1}{2L} \ln \frac{1 + \cos \theta + M^2/s}{M^2/s} & \text{for } \cos \theta < 0 \\ 1 - \frac{1}{2L} \ln \frac{1 - \cos \theta + M^2/s}{M^2/s} & \text{for } \cos \theta \geq 0 \end{cases} \tag{11.60}$$

The phase-space factor:

$$f(x) = \begin{cases} \frac{M^2}{s} \exp(xL) L & \text{for } x < \frac{1}{2} \\ \frac{M^2}{s} \exp((1-x)L) L & \text{for } x \geq \frac{1}{2} \end{cases} \tag{11.61}$$

⟨Constants for ct pole mapping⟩≡
```
if (map%variable_limits .or. map%b_unknown) then
```

```
      map%b1 = map%mass**2 / s
      map%b2 = log ((map%b1 + 1) / map%b1)
      map%b3 = 0
      map%b_unknown = .false.
   end if
```

⟨*Apply ct pole mapping*⟩≡
```
   if (x < .5_default) then
      ct1 = map%b1 * exp (2 * x * map%b2)
      ct = ct1 - map%b1 - 1
   else
      ct1 = map%b1 * exp (2 * (1-x) * map%b2)
      ct = -(ct1 - map%b1) + 1
   end if
   if (ct >= -1 .and. ct <= 1) then
      st = sqrt (1 - ct**2)
      f = ct1 * map%b2
   else
      ct = 1;  st = 0;  f = 0
   end if
```

⟨*Apply inverse ct pole mapping*⟩≡
```
   if (ct < 0) then
      ct1 = ct + map%b1 + 1
      x = log (ct1 / map%b1) / (2 * map%b2)
   else
      ct1 = -ct + map%b1 + 1
      x = 1 - log (ct1 / map%b1) / (2 * map%b2)
   end if
   f = ct1 * map%b2
```

11.5 Phase-space trees

The phase space evaluation is organized in terms of trees, where each branch corresponds to three integrations: m^2, $\cos\theta$, and ϕ. The complete tree thus makes up a specific parameterization of the multidimensional phase-space integral. For the multi-channel integration, the phase-space tree is a single channel.

The trees imply mappings of formal Feynman tree graphs into arrays of integer numbers: Each branch, corresponding to a particular line in the graph, is assigned an integer code c (with kind value $\mathtt{TC}$ = tree code).

In this integer, each bit determines whether a particular external momentum flows through the line. The external branches therefore have codes $1, 2, 4, 8, \ldots$. An internal branch has those bits ORed corresponding to the momenta flowing through it. For example, a branch with momentum $p_1 + p_4$ has code $2^0 + 2^3 = 1 + 8 = 9$.

There is a two-fold ambiguity: Momentum conservation implies that the branch with code

$$c_0 = \sum_{i=1}^{n(\text{ext})} 2^{i-1} \tag{11.62}$$

i.e. the branch with momentum $p_1 + p_2 + \ldots p_n$ has momentum zero, which is equivalent to tree code 0 by definition. Correspondingly,

$$c \quad \text{and} \quad c_0 - c = c \text{ XOR } c_0 \tag{11.63}$$

are equivalent. E.g., if there are five externals with codes $c = 1, 2, 4, 8, 16$, then $c = 9$ and $\bar{c} = 31 - 9 = 22$ are equivalent.

This ambiguity may be used to assign a direction to the line: If all momenta are understood as outgoing, $c = 9$ in the example above means $p_1 + p_4$, but $c = 22$ means $p_2 + p_3 + p_5 = -(p_1 + p_4)$.

Here we make use of the ambiguity in a slightly different way. First, the initial particles are singled out as those externals with the highest bits, the IN-bits. (Here: 8 and 16 for a $2 \to 3$ scattering process, 16 only for a $1 \to 4$ decay.) Then we invert those codes where all IN-bits are set. For a decay process this maps each tree of an equivalence class onto a unique representative (that one with the smallest integer codes). For a scattering process we proceed further:

The ambiguity remains in all branches where only one IN-bit is set, including the initial particles. If there are only externals with this property, we have an s-channel graph which we leave as it is. In all other cases, an internal with only one IN-bit is a t-channel line, which for phase space integration should be associated with one of the initial momenta as a reference axis. We take that one whose bit is set in the current tree code. (E.g., for branch $c = 9$ we use the initial particle $c = 8$ as reference axis, whereas for the same branch we would take $c = 16$ if it had been assigned $\bar{c} = 31 - 9 = 22$ as tree code.) Thus, different ways of coding the same t-channel graph imply different phase space parameterizations.

s-channel graphs have a unique parameterization. The same sets of parameterizations are used for t-channel graphs, except for the reference frames of their angular parts. We map each t-channel graph onto an s-channel graph as follows:

Working in ascending order, for each t-channel line (whose code has exactly one IN-bit set) the attached initial line is flipped upstream, while the outgoing

line is flipped downstream. (This works only if *t*-channel graphs are always parameterized beginning at their outer vertices, which we require as a restriction.) After all possible flips have been applied, we have an *s*-channel graph. We only have to remember the initial particle a vertex was originally attached to.

⟨phs_trees.f90⟩≡
  ⟨*File header*⟩

  ```
  module phs_trees
  ```

  ⟨*Use kinds*⟩
  ```
    use kinds, only: TC !NODEP!
  ```
  ⟨*Use strings*⟩
  ```
    use constants, only: twopi, twopi2, twopi5 !NODEP!
  ```
  ⟨*Use file utils*⟩
  ```
    use limits, only: FMT_19 !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use permutations, only: permutation_t, permutation_size
    use permutations, only: permutation_init, permutation_find
    use permutations, only: tc_decay_level, tc_permute
    use models
    use flavors
    use mappings
  ```

  ⟨*Standard module head*⟩

  ⟨*PHS trees: public*⟩

  ⟨*PHS trees: types*⟩

  ```
  contains
  ```

  ⟨*PHS trees: procedures*⟩

  ```
  end module phs_trees
  ```

11.5.1 Particles

We define a particle type which contains only four-momentum and invariant mass squared, and a flag that tells whether the momentum is filled or not.

⟨*PHS trees: public*⟩≡
  ```
    public :: phs_prt_t
  ```

⟨*PHS trees: types*⟩≡
  ```
    type :: phs_prt_t
       private
       logical :: defined = .false.
       type(vector4_t) :: p
       real(default) :: p2
    end type phs_prt_t
  ```

Set contents:
⟨*PHS trees: public*⟩+≡

```
    public :: phs_prt_set_defined
    public :: phs_prt_set_undefined
    public :: phs_prt_set_momentum
    public :: phs_prt_set_msq
```

⟨*PHS trees: procedures*⟩≡
```
    elemental subroutine phs_prt_set_defined (prt)
      type(phs_prt_t), intent(inout) :: prt
      prt%defined = .true.
    end subroutine phs_prt_set_defined

    elemental subroutine phs_prt_set_undefined (prt)
      type(phs_prt_t), intent(inout) :: prt
      prt%defined = .false.
    end subroutine phs_prt_set_undefined

    elemental subroutine phs_prt_set_momentum (prt, p)
      type(phs_prt_t), intent(inout) :: prt
      type(vector4_t), intent(in) :: p
      prt%p = p
    end subroutine phs_prt_set_momentum

    elemental subroutine phs_prt_set_msq (prt, p2)
      type(phs_prt_t), intent(inout) :: prt
      real(default), intent(in) :: p2
      prt%p2 = p2
    end subroutine phs_prt_set_msq
```

Access methods:

⟨*PHS trees: public*⟩+≡
```
    public :: phs_prt_is_defined
    public :: phs_prt_get_momentum
    public :: phs_prt_get_msq
```

⟨*PHS trees: procedures*⟩+≡
```
    elemental function phs_prt_is_defined (prt) result (defined)
      logical :: defined
      type(phs_prt_t), intent(in) :: prt
      defined = prt%defined
    end function phs_prt_is_defined

    elemental function phs_prt_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(phs_prt_t), intent(in) :: prt
      p = prt%p
    end function phs_prt_get_momentum

    elemental function phs_prt_get_msq (prt) result (p2)
      real(default) :: p2
      type(phs_prt_t), intent(in) :: prt
      p2 = prt%p2
    end function phs_prt_get_msq
```

Addition of momenta (invariant mass square is computed).

⟨*PHS trees: public*⟩+≡
```
    public :: phs_prt_combine
```

⟨*PHS trees: procedures*⟩+≡
```
    elemental subroutine phs_prt_combine (prt, prt1, prt2)
      type(phs_prt_t), intent(inout) :: prt
      type(phs_prt_t), intent(in) :: prt1, prt2
      prt%defined = .true.
      prt%p = prt1%p + prt2%p
      prt%p2 = prt%p ** 2
    end subroutine phs_prt_combine
```

Output

⟨*PHS trees: public*⟩+≡
```
    public :: phs_prt_write
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_prt_write (prt, unit)
      type(phs_prt_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      if (prt%defined) then
         call vector4_write (prt%p, u)
         write (u, "(1x,A,1x," // FMT_19 // ")") "T = ", prt%p2
      else
         write (u, "(3x,A)") "[undefined]"
      end if
    end subroutine phs_prt_write
```

11.5.2   The phase-space tree type

Definition

In the concrete implementation, each branch c may have two *daughters* c_1 and c_2 such that $c_1 + c_2 = c$, a *sibling* c_s and a *mother* c_m such that $c + c_s = c_m$, and a *friend* which is kept during flips, such that it can indicate a fixed reference frame. Absent entries are set $c = 0$.

First, declare the branch type. There is some need to have this public. Give initializations for all components, so no `init` routine is necessary. The branch has some information about the associated coordinates and about connections.

⟨*PHS trees: types*⟩+≡
```
    type :: phs_branch_t
      private
      logical :: set = .false.
      logical :: inverted_decay = .false.
      logical :: inverted_axis = .false.
      integer(TC) :: mother = 0
      integer(TC) :: sibling = 0
      integer(TC) :: friend = 0
      integer(TC) :: origin = 0
```

```
      integer(TC), dimension(2) :: daughter = 0
      integer :: firstborn = 0
      logical :: has_children = .false.
      logical :: has_friend = .false.
    end type phs_branch_t
```

The tree type: No initialization, this is done by `phs_tree_init`. In addition to the branch array which

The branches are collected in an array which holds all possible branches, of which only a few are set. After flips have been applied, the branch $c_M = \sum_{i=1}^{n(\text{fin})} 2^{i-1}$ must be there, indicating the mother of all decay products. In addition, we should check for consistency at the beginning.

`n_branches` is the number of those actually set. `n_externals` defines the number of significant bit, and `mask` is a code where all bits are set. Analogous: `n_in` and `mask_in` for the incoming particles.

The `mapping` array contains the mappings associated to the branches (corresponding indices). The array `mass_sum` contains the sum of the real masses of the external final-state particles associated to the branch. During phase-space evaluation, this determines the boundaries.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_t
```

⟨*PHS trees: types*⟩+≡
```
    type :: phs_tree_t
      private
      integer :: n_branches, n_externals, n_in, n_msq, n_angles
      integer(TC) :: n_branches_tot, n_branches_out
      integer(TC) :: mask, mask_in, mask_out
      type(phs_branch_t), dimension(:), allocatable :: branch
      type(mapping_t), dimension(:), allocatable :: mapping
      real(default), dimension(:), allocatable :: mass_sum
      real(default), dimension(:), allocatable :: effective_mass
      real(default), dimension(:), allocatable :: effective_width
    end type phs_tree_t
```

The maximum number of external particles that can be represented is related to the bit size of the integer that stores binary codes. With the default integer of 32 bit on common machines, this is more than enough space. If `TC` is actually the default integer kind, there is no need to keep it separate, but doing so marks this as a special type of integer. So, just state that the maximum number is 32:

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: MAX_EXTERNAL = 32
```

Constructor and destructor

Allocate memory for a phase-space tree with given number of externals and incoming. The number of allocated branches can easily become large, but appears manageable for realistic cases, e.g., for `n_in=2` and `n_out=8` we get $2^{10} - 1 = 1023$.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_init
    public :: phs_tree_final
```

1429

Here we set the masks for incoming and for all externals.

⟨*PHS trees: procedures*⟩+≡
```
elemental subroutine phs_tree_init (tree, n_in, n_out, n_masses, n_angles)
  type(phs_tree_t), intent(inout) :: tree
  integer, intent(in) :: n_in, n_out, n_masses, n_angles
  integer(TC) :: i
  tree%n_externals = n_in + n_out
  tree%n_branches_tot = 2**(n_in+n_out) - 1
  tree%n_branches_out = 2**n_out - 1
  tree%mask = 0
  do i = 0, n_in + n_out - 1
     tree%mask = ibset (tree%mask, i)
  end do
  tree%n_in = n_in
  tree%mask_in = 0
  do i = n_out, n_in + n_out - 1
     tree%mask_in = ibset (tree%mask_in, i)
  end do
  tree%mask_out = ieor (tree%mask, tree%mask_in)
  tree%n_msq = n_masses
  tree%n_angles = n_angles
  allocate (tree%branch (tree%n_branches_tot))
  tree%n_branches  = 0
  allocate (tree%mapping (tree%n_branches_out))
  allocate (tree%mass_sum (tree%n_branches_out))
  allocate (tree%effective_mass (tree%n_branches_out))
  allocate (tree%effective_width (tree%n_branches_out))
end subroutine phs_tree_init

elemental subroutine phs_tree_final (tree)
  type(phs_tree_t), intent(inout) :: tree
  deallocate (tree%branch)
  deallocate (tree%mapping)
  deallocate (tree%mass_sum)
  deallocate (tree%effective_mass)
  deallocate (tree%effective_width)
end subroutine phs_tree_final
```

Screen output

Write only the branches that are set:

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_write
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_write (tree, unit)
  type(phs_tree_t), intent(in) :: tree
  integer, intent(in), optional :: unit
  integer :: u
  integer(TC) :: k
  u = output_unit (unit);  if (u < 0)  return
  write (u, '(3X,A,1x,I0,5X,A,I3)') &
       'External:', tree%n_externals, 'Mask:', tree%mask
```

```fortran
    write (u, '(3X,A,1x,I0,5X,A,I3)') &
         'Incoming:', tree%n_in, 'Mask:', tree%mask_in
    write (u, '(3X,A,1x,I0,5X,A,I3)') &
         'Branches:', tree%n_branches
    do k = size (tree%branch), 1, -1
       if (tree%branch(k)%set) &
            call phs_branch_write (tree%branch(k), unit=unit, kval=k)
    end do
    do k = 1, size (tree%mapping)
       call mapping_write (tree%mapping (k), unit, verbose=.true.)
    end do
    write (u, "(3x,A)") "Arrays: mass_sum, effective_mass, effective_width"
    do k = 1, size (tree%mass_sum)
       if (tree%branch(k)%set) then
          write (u, "(5x,I0,3(2x," // FMT_19 // "))") k, tree%mass_sum(k), &
               tree%effective_mass(k), tree%effective_width(k)
       end if
    end do
  end subroutine phs_tree_write

  subroutine phs_branch_write (b, unit, kval)
    type(phs_branch_t), intent(in) :: b
    integer, intent(in), optional :: unit
    integer(TC), intent(in), optional :: kval
    integer :: u
    integer(TC) :: k
    character(len=6) :: tmp
    character(len=1) :: firstborn(2), sign_decay, sign_axis
    integer :: i
    u = output_unit (unit);  if (u < 0)  return
    k = 0;  if (present (kval))  k = kval
    if (b%origin /= 0) then
       write(tmp, '(A,I4,A)') '(', b%origin, ')'
    else
       tmp = ' '
    end if
    do i=1, 2
       if (b%firstborn == i) then
          firstborn(i) = "*"
       else
          firstborn(i) = " "
       end if
    end do
    if (b%inverted_decay) then
       sign_decay = "-"
    else
       sign_decay = "+"
    end if
    if (b%inverted_axis) then
       sign_axis = "-"
    else
       sign_axis = "+"
    end if
    if (b%has_children) then
```

```
                 if (b%has_friend) then
                    write(u,'(4X,A1,I0,3x,A,1X,A,I0,A1,1x,I0,A1,1X,A1,1X,A,1x,I0)') &
                         &   '*', k, tmp, &
                         &   'Daughters: ', &
                         &   b%daughter(1), firstborn(1), &
                         &   b%daughter(2), firstborn(2), sign_decay, &
                         &   'Friend:     ', b%friend
                 else
                    write(u,'(4X,A1,I0,3x,A,1X,A,I0,A1,1x,I0,A1,1X,A1,1X,A)') &
                         &   '*', k, tmp, &
                         &   'Daughters: ', &
                         &   b%daughter(1), firstborn(1), &
                         &   b%daughter(2), firstborn(2), sign_decay, &
                         &   '(axis '//sign_axis//')'
                 end if
              else
                 write(u,'(5X,I0)') k
              end if
          end subroutine phs_branch_write
```

11.5.3 PHS tree setup

Transformation into an array of branch codes and back

Assume that the tree/array has been created before with the appropriate length
and is empty.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_from_array
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_from_array (tree, a)
      type(phs_tree_t), intent(inout) :: tree
      integer(TC), dimension(:), intent(in) :: a
      integer :: i
      integer(TC) :: k
```
⟨*Set branches from array* a⟩
⟨*Set external branches if necessary*⟩
⟨*Check number of branches*⟩
⟨*Determine the connections*⟩
```
    contains
```
⟨*Subroutine: set relatives*⟩
```
    end subroutine phs_tree_from_array
```

First, set all branches specified by the user. If all IN-bits are set, we invert the
branch code.

⟨*Set branches from array* a⟩≡
```
    do i=1, size(a)
       k = a(i)
       if (iand(k, tree%mask_in) == tree%mask_in)  k = ieor(tree%mask, k)
       tree%branch(k)%set = .true.
       tree%n_branches = tree%n_branches+1
    end do
```

1432

The external branches are understood, so set them now if not yet done. In all cases ensure that the representative with one bit set is used, except for decays where the in-particle is represented by all OUT-bits set instead.

⟨*Set external branches if necessary*⟩≡
```
do i=0, tree%n_externals-1
   k = ibset(0,i)
   if (iand(k, tree%mask_in) == tree%mask_in)  k = ieor(tree%mask, k)
   if (tree%branch(ieor(tree%mask, k))%set) then
      tree%branch(ieor(tree%mask, k))%set = .false.
      tree%branch(k)%set = .true.
   else if (.not.tree%branch(k)%set) then
      tree%branch(k)%set = .true.
      tree%n_branches = tree%n_branches+1
   end if
end do
```

Now the number of branches set can be checked. Here we assume that the tree is binary. For three externals there are three branches in total, and for each additional external branch we get another internal one.

⟨*Check number of branches*⟩≡
```
if (tree%n_branches /= tree%n_externals*2-3) then
   call phs_tree_write (tree)
   call msg_bug &
        & (" Wrong number of branches set in phase space tree")
end if
```

For all branches that are set, except for the externals, we try to find the daughter branches:

⟨*Determine the connections*⟩≡
```
do k=1, size (tree%branch)
   if (tree%branch(k)%set .and. tc_decay_level (k) /= 1) then
      call branch_set_relatives(k)
   end if
end do
```

To this end, we scan all codes less than the current code, whether we can find two branches which are set and which together give the current code. After that, the tree may still not be connected, but at least we know if a branch does not have daughters: This indicates some inconsistency.

The algorithm ensures that, at this stage, the first daughter has a smaller code value than the second one.

⟨*Subroutine: set relatives*⟩≡
```
subroutine branch_set_relatives (k)
  integer(TC), intent(in) :: k
  integer(TC) :: m,n
  do m=1, k-1
     if(iand(k,m)==m) then
        n = ieor(k,m)
        if ( tree%branch(m)%set .and. tree%branch(n)%set ) then
           tree%branch(k)%daughter(1) = m;  tree%branch(k)%daughter(2) = n
           tree%branch(m)%mother      = k;  tree%branch(n)%mother      = k
           tree%branch(m)%sibling     = n;  tree%branch(n)%sibling     = m
           tree%branch(k)%has_children = .true.
           return
```

```
          end if
        end if
      end do
      call phs_tree_write (tree)
      call msg_bug &
          & (" Missing daughter branch(es) in phase space tree")
    end subroutine branch_set_relatives
```

The inverse: this is trivial, fortunately.

Flip *t*-channel into *s*-channel

Flipping the tree is done upwards, beginning from the decay products. First we select a *t*-channel branch k: one which is set, which does have an IN-bit, and which is not an external particle.

Next, we determine the adjacent in-particle (called the 'friend' f here, since it will provide the reference axis for the angular integration). In addition, we look for the 'mother' and 'sibling' of this particle. If the latter field is empty, we select the (unique) other out-particle which has no mother, calling the internal subroutine `find_orphan`.

The flip is done as follows: We assume that the first daughter d is an *s*-channel line, which is true if the daughters are sorted. This will stay the first daughter. The second one is a *t*-channel line; it is exchanged with the 'sibling' s. The new line which replaces the branch k is just the sum of s and d. In addition, we have to rearrange the relatives of s and d, as well of f.

Finally, we flip 'sibling' and 'friend' and set the new *s*-channel branch n which replaces the *t*-channel branch k. After this is complete, we are ready to execute another flip.

[Although the friend is not needed for the final flip, since it would be an initial particle anyway, we need to know whether we have *t*- or *u*-channel.]

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_flip_t_to_s_channel
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_flip_t_to_s_channel (tree)
      type(phs_tree_t), intent(inout) :: tree
      integer(TC) :: k, f, m, n, d, s
      if (tree%n_in == 2) then
         FLIP: do k=3, tree%mask-1
            if (.not. tree%branch(k)%set) cycle FLIP
            f = iand(k,tree%mask_in)
            if (f==0 .or. f==k) cycle FLIP
            m = tree%branch(k)%mother
            s = tree%branch(k)%sibling
            if (s==0) call find_orphan(s)
            d = tree%branch(k)%daughter(1)
            n = ior(d,s)
            tree%branch(k)%set = .false.
            tree%branch(n)%set = .true.
            tree%branch(n)%origin = k
            tree%branch(n)%daughter(1) = d; tree%branch(d)%mother  = n
            tree%branch(n)%daughter(2) = s; tree%branch(s)%mother  = n
```

```
                tree%branch(n)%has_children = .true.
                tree%branch(d)%sibling = s;   tree%branch(s)%sibling = d
                tree%branch(n)%sibling = f;   tree%branch(f)%sibling = n
                tree%branch(n)%mother      = m
                tree%branch(f)%mother      = m
                if (m/=0) then
                    tree%branch(m)%daughter(1) = n
                    tree%branch(m)%daughter(2) = f
                end if
                tree%branch(n)%friend = f
                tree%branch(n)%has_friend = .true.
                tree%branch(n)%firstborn = 2
            end do FLIP
        end if
    contains
        subroutine find_orphan(s)
            integer(TC) :: s
            do s=1, tree%mask_out
                if (tree%branch(s)%set .and. tree%branch(s)%mother==0) return
            end do
            call phs_tree_write (tree)
            call msg_bug (" Can't flip phase space tree to channel")
        end subroutine find_orphan
    end subroutine phs_tree_flip_t_to_s_channel
```

After the tree has been flipped, one may need to determine what has become
of a particular *t*-channel branch. This function gives the bincode of the flipped
tree. If the original bincode does not contain IN-bits, we leave it as it is.

⟨*PHS trees: procedures*⟩+≡
```
    function tc_flipped (tree, kt) result (ks)
      type(phs_tree_t), intent(in) :: tree
      integer(TC), intent(in) :: kt
      integer(TC) :: ks
      if (iand (kt, tree%mask_in) == 0) then
          ks = kt
      else
          ks = tree%branch(iand (kt, tree%mask_out))%mother
      end if
    end function tc_flipped
```

Scan a tree and make sure that the first daughter has always a smaller code
than the second one. Furthermore, delete any `friend` entry in the root branch
– this branching has the incoming particle direction as axis anyway. Keep track
of reordering by updating `inverted_axis`, `inverted_decay` and `firstborn`.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_canonicalize
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_canonicalize (tree)
      type(phs_tree_t), intent(inout) :: tree
      integer :: n_out
      integer(TC) :: k_out
      call branch_canonicalize (tree%branch(tree%mask_out))
```

```
      n_out = tree%n_externals - tree%n_in
      k_out = tree%mask_out
      if (tree%branch(k_out)%has_friend &
            & .and. tree%branch(k_out)%friend == ibset (0, n_out)) then
         tree%branch(k_out)%inverted_axis = .not.tree%branch(k_out)%inverted_axis
      end if
      tree%branch(k_out)%has_friend = .false.
      tree%branch(k_out)%friend = 0
    contains
      recursive subroutine branch_canonicalize (b)
        type(phs_branch_t), intent(inout) :: b
        integer(TC) :: d1, d2
        if (b%has_children) then
            d1 = b%daughter(1)
            d2 = b%daughter(2)
            if (d1 > d2) then
               b%daughter(1) = d2
               b%daughter(2) = d1
               b%inverted_decay = .not.b%inverted_decay
               if (b%firstborn /= 0)  b%firstborn = 3 - b%firstborn
            end if
            call branch_canonicalize (tree%branch(b%daughter(1)))
            call branch_canonicalize (tree%branch(b%daughter(2)))
        end if
      end subroutine branch_canonicalize
    end subroutine phs_tree_canonicalize
```

Mappings

Initialize a mapping for the current tree. This is done while reading from file, so the mapping parameters are read, but applied to the flipped tree. Thus, the size of the array of mappings is given by the number of outgoing particles only.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_init_mapping
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_init_mapping (tree, k, type, pdg, model)
      type(phs_tree_t), intent(inout) :: tree
      integer(TC), intent(in) :: k
      type(string_t), intent(in) :: type
      integer, intent(in) :: pdg
      type(model_t), intent(in), target :: model
      integer(TC) :: kk
      kk = tc_flipped (tree, k)
      call mapping_init (tree%mapping(kk), kk, type, pdg, model)
    end subroutine phs_tree_init_mapping
```

Set the physical parameters for the mapping, using a specific parameter set. Also set the mass sum array.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_set_mapping_parameters
```

⟨PHS trees: procedures⟩+≡

```
subroutine phs_tree_set_mapping_parameters &
     (tree, mapping_defaults, variable_limits)
  type(phs_tree_t), intent(inout) :: tree
  type(mapping_defaults_t), intent(in) :: mapping_defaults
  logical, intent(in) :: variable_limits
  integer(TC) :: k
  do k = 1, tree%n_branches_out
     call mapping_set_parameters &
          (tree%mapping(k), mapping_defaults, variable_limits)
  end do
end subroutine phs_tree_set_mapping_parameters
```

Return the mapping for the sum of all outgoing particles. This should either be no mapping or a global s-channel mapping.

⟨PHS trees: public⟩+≡

```
public :: phs_tree_assign_s_mapping
```

⟨PHS trees: procedures⟩+≡

```
subroutine phs_tree_assign_s_mapping (tree, mapping)
  type(phs_tree_t), intent(in) :: tree
  type(mapping_t), intent(out) :: mapping
  mapping = tree%mapping(tree%mask_out)
end subroutine phs_tree_assign_s_mapping
```

Kinematics

Fill the mass sum array, starting from the external particles and working down to the tree root. For each bincode `k` we scan the bits in `k`; if only one is set, we take the physical mass of the corresponding external particle; if more then one is set, we sum up the two masses (which we know have already been set).

⟨PHS trees: public⟩+≡

```
public :: phs_tree_set_mass_sum
```

⟨PHS trees: procedures⟩+≡

```
subroutine phs_tree_set_mass_sum (tree, flv)
  type(phs_tree_t), intent(inout) :: tree
  type(flavor_t), dimension(:), intent(in) :: flv
  integer(TC) :: k
  integer :: i
  tree%mass_sum = 0
  do k = 1, tree%n_branches_out
     do i = 0, size (flv) - 1
        if (btest(k,i)) then
           if (ibclr(k,i) == 0) then
              tree%mass_sum(k) = flavor_get_mass (flv(i+1))
           else
              tree%mass_sum(k) = &
                    tree%mass_sum(ibclr(k,i)) + tree%mass_sum(ibset(0,i))
           end if
           exit
        end if
```

1437

```
         end do
      end do
    end subroutine phs_tree_set_mass_sum
```

Set the effective masses and widths. For each non-resonant branch in a tree, the effective mass is equal to the sum of the effective masses of the children (and analogous for the width). External particles have their real mass and width zero. For resonant branches, we insert mass and width from the corresponding mapping.

This routine has `phs_tree_set_mass_sum` and `phs_tree_set_mapping_parameters` as prerequisites.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_set_effective_masses
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_set_effective_masses (tree)
      type(phs_tree_t), intent(inout) :: tree
      tree%effective_mass = 0
      tree%effective_width = 0
      call set_masses_x (tree%mask_out)
    contains
      recursive subroutine set_masses_x (k)
        integer(TC), intent(in) :: k
        integer(TC) :: k1, k2
        if (tree%branch(k)%has_children) then
           k1 = tree%branch(k)%daughter(1)
           k2 = tree%branch(k)%daughter(2)
           call set_masses_x (k1)
           call set_masses_x (k2)
           if (mapping_is_s_channel (tree%mapping(k))) then
              tree%effective_mass(k) = mapping_get_mass (tree%mapping(k))
              tree%effective_width(k) = mapping_get_width (tree%mapping(k))
           else
              tree%effective_mass(k) = &
                   tree%effective_mass(k1) + tree%effective_mass(k2)
              tree%effective_width(k) = &
                   tree%effective_width(k1) + tree%effective_width(k2)
           end if
        else
           tree%effective_mass(k) = tree%mass_sum(k)
        end if
      end subroutine set_masses_x
    end subroutine phs_tree_set_effective_masses
```

Define step mappings, recursively, for the decay products of all intermediate resonances. Step mappings account for the fact that a branch may originate from a resonance, which almost replaces the upper limit on the possible invariant mass. The step mapping implements a smooth cutoff that interpolates between the resonance and the real kinematic limit. The mapping width determines the sharpness of the cutoff.

Step mappings are inserted only for branches that are not mapped otherwise.

At each branch, we record the mass that is effectively available for phase

space, by taking the previous limit and subtracting the effective mass of the sibling branch. Widths are added, not subtracted.

If we encounter a resonance decay, we discard the previous limit and replace it by the mass and width of the resonance, also subtracting the sibling branch.

Initially, the limit is zero, so it becomes negative at any branch. Only if there is a resonance, the limit becomes positive. Whenever the limit is positive, and the current branch decays, we activate a step mapping for the current branch.

As a result, step mappings are implemented for all internal lines that originate from an intermediate resonance decay.

The flag `variable_limits` applies to the ultimate limit from the available energy, not to the intermediate resonances whose masses are always fixed.

This routine requires `phs_tree_set_effective_masses`

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_set_step_mappings
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_set_step_mappings (tree, exp_type, variable_limits)
      type(phs_tree_t), intent(inout) :: tree
      logical, intent(in) :: exp_type
      logical, intent(in) :: variable_limits
      type(string_t) :: map_str
      integer(TC) :: k
      if (exp_type) then
         map_str = "step_exp"
      else
         map_str = "step_hyp"
      end if
      k = tree%mask_out
      call set_step_mappings_x (k, 0._default, 0._default)
    contains
      recursive subroutine set_step_mappings_x (k, m_limit, w_limit)
        integer(TC), intent(in) :: k
        real(default), intent(in) :: m_limit, w_limit
        integer(TC), dimension(2) :: kk
        real(default), dimension(2) :: m, w
        if (tree%branch(k)%has_children) then
           if (m_limit > 0) then
              if (.not. mapping_is_set (tree%mapping(k))) then
                 call mapping_init (tree%mapping(k), k, map_str)
                 call mapping_set_step_mapping_parameters (tree%mapping(k), &
                      m_limit, w_limit, &
                      variable_limits)
              end if
           end if
           kk = tree%branch(k)%daughter
           m = tree%effective_mass(kk)
           w = tree%effective_width(kk)
           if (mapping_is_s_channel (tree%mapping(k))) then
              call set_step_mappings_x (kk(1), &
                   mapping_get_mass (tree%mapping(k)) - m(2), &
                   mapping_get_width (tree%mapping(k)) + w(2))
              call set_step_mappings_x (kk(2), &
                   mapping_get_mass (tree%mapping(k)) - m(1), &
```

```
                mapping_get_width (tree%mapping(k)) + w(1))
        else if (m_limit > 0) then
           call set_step_mappings_x (kk(1), &
                m_limit - m(2), &
                w_limit + w(2))
           call set_step_mappings_x (kk(2), &
                m_limit - m(1), &
                w_limit + w(1))
        else
           call set_step_mappings_x (kk(1), &
                - m(2), &
                + w(2))
           call set_step_mappings_x (kk(2), &
                - m(1), &
                + w(1))
        end if
     end if
   end subroutine set_step_mappings_x
 end subroutine phs_tree_set_step_mappings
```

Structural comparison

This function allows to check whether one tree is the permutation of another one. The permutation is applied to the second tree in the argument list. We do not make up a temporary permuted tree, but compare the two trees directly. The branches are scanned recursively, where for each daughter we check the friend and the mapping as well. Once a discrepancy is found, the recursion is exited immediately.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_equivalent
```

⟨*PHS trees: procedures*⟩+≡
```
    function phs_tree_equivalent (t1, t2, perm) result (is_equal)
      type(phs_tree_t), intent(in) :: t1, t2
      type(permutation_t), intent(in) :: perm
      logical :: equal, is_equal
      integer(TC) :: k1, k2, mask_in
      k1 = t1%mask_out
      k2 = t2%mask_out
      mask_in = t1%mask_in
      equal = .true.
      call check (t1%branch(k1), t2%branch(k2), k1, k2)
      is_equal = equal
    contains
      recursive subroutine check (b1, b2, k1, k2)
        type(phs_branch_t), intent(in) :: b1, b2
        integer(TC), intent(in) :: k1, k2
        integer(TC), dimension(2) :: d1, d2, pd2
        integer :: i
        if (.not.b1%has_friend .and. .not.b2%has_friend) then
           equal = .true.
        else if (b1%has_friend .and. b2%has_friend) then
           equal = (b1%friend == tc_permute (b2%friend, perm, mask_in))
```

```
          end if
          if (equal) then
              if (b1%has_children .and. b2%has_children) then
                  d1 = b1%daughter
                  d2 = b2%daughter
                  do i=1, 2
                      pd2(i) = tc_permute (d2(i), perm, mask_in)
                  end do
                  if (d1(1)==pd2(1) .and. d1(2)==pd2(2)) then
                      equal = (b1%firstborn == b2%firstborn)
                      if (equal) call check &
                          &      (t1%branch(d1(1)), t2%branch(d2(1)), d1(1), d2(1))
                      if (equal) call check &
                          &      (t1%branch(d1(2)), t2%branch(d2(2)), d1(2), d2(2))
                  else if (d1(1)==pd2(2) .and. d1(2)==pd2(1)) then
                      equal = ( (b1%firstborn == 0 .and. b2%firstborn == 0) &
                          &    .or. (b1%firstborn == 3 - b2%firstborn) )
                      if (equal) call check &
                          &      (t1%branch(d1(1)), t2%branch(d2(2)), d1(1), d2(2))
                      if (equal) call check &
                          &      (t1%branch(d1(2)), t2%branch(d2(1)), d1(2), d2(1))
                  else
                      equal = .false.
                  end if
              end if
          end if
          if (equal) then
              equal = (t1%mapping(k1) == t2%mapping(k2))
          end if
      end subroutine check
  end function phs_tree_equivalent
```

Scan two decay trees and determine the correspondence of mass variables, i.e.,
the permutation that transfers the ordered list of mass variables belonging to
the second tree into the first one. Mass variables are assigned beginning from
branches and ending at the root.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_find_msq_permutation
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_find_msq_permutation (tree1, tree2, perm2, msq_perm)
      type(phs_tree_t), intent(in) :: tree1, tree2
      type(permutation_t), intent(in) :: perm2
      type(permutation_t), intent(out) :: msq_perm
      type(permutation_t) :: perm1
      integer(TC) :: mask_in, root
      integer(TC), dimension(:), allocatable :: index1, index2
      integer :: i
      allocate (index1 (tree1%n_msq), index2 (tree2%n_msq))
      call permutation_init (perm1, permutation_size (perm2))
      mask_in = tree1%mask_in
      root = tree1%mask_out
      i = 0
      call tree_scan (tree1, root, perm1, index1)
```

```
      i = 0
      call tree_scan (tree2, root, perm2, index2)
      call permutation_find (msq_perm, index1, index2)
    contains
      recursive subroutine tree_scan (tree, k, perm, index)
        type(phs_tree_t), intent(in) :: tree
        integer(TC), intent(in) :: k
        type(permutation_t), intent(in) :: perm
        integer, dimension(:), intent(inout) :: index
        if (tree%branch(k)%has_children) then
            call tree_scan (tree, tree%branch(k)%daughter(1), perm, index)
            call tree_scan (tree, tree%branch(k)%daughter(2), perm, index)
            i = i + 1
            if (i <= size (index))  index(i) = tc_permute (k, perm, mask_in)
        end if
      end subroutine tree_scan
    end subroutine phs_tree_find_msq_permutation
```

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_find_angle_permutation
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_find_angle_permutation &
         (tree1, tree2, perm2, angle_perm, sig2)
      type(phs_tree_t), intent(in) :: tree1, tree2
      type(permutation_t), intent(in) :: perm2
      type(permutation_t), intent(out) :: angle_perm
      logical, dimension(:), allocatable, intent(out) :: sig2
      type(permutation_t) :: perm1
      integer(TC) :: mask_in, root
      integer(TC), dimension(:), allocatable :: index1, index2
      logical, dimension(:), allocatable :: sig1
      integer :: i
      allocate (index1 (tree1%n_angles), index2 (tree2%n_angles))
      allocate (sig1 (tree1%n_angles), sig2 (tree2%n_angles))
      call permutation_init (perm1, permutation_size (perm2))
      mask_in = tree1%mask_in
      root = tree1%mask_out
      i = 0
      call tree_scan (tree1, root, perm1, index1, sig1)
      i = 0
      call tree_scan (tree2, root, perm2, index2, sig2)
      call permutation_find (angle_perm, index1, index2)
    contains
      recursive subroutine tree_scan (tree, k, perm, index, sig)
        type(phs_tree_t), intent(in) :: tree
        integer(TC), intent(in) :: k
        type(permutation_t), intent(in) :: perm
        integer, dimension(:), intent(inout) :: index
        logical, dimension(:), intent(inout) :: sig
        integer(TC) :: k1, k2, kp
        logical :: s
        if (tree%branch(k)%has_children) then
            k1 = tree%branch(k)%daughter(1)
```

```
      k2 = tree%branch(k)%daughter(2)
      s = (tc_permute(k1, perm, mask_in) < tc_permute(k2, perm, mask_in))
      kp = tc_permute (k, perm, mask_in)
      i = i + 1
      index(i) = kp
      sig(i) = s
      i = i + 1
      index(i) = - kp
      sig(i) = s
      call tree_scan (tree, k1, perm, index, sig)
      call tree_scan (tree, k2, perm, index, sig)
    end if
  end subroutine tree_scan
end subroutine phs_tree_find_angle_permutation
```

11.5.4 Phase-space evaluation

Phase-space volume

We compute the phase-space volume recursively, following the same path as for computing other kinematical variables. However, the volume depends just on $\sqrt{\hat{s}}$, not on the momentum configuration.

   Note: counting branches, we may replace this by a simple formula.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_compute_volume
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_compute_volume (tree, sqrts, volume)
      type(phs_tree_t), intent(in) :: tree
      real(default), intent(in) :: sqrts
      real(default), intent(out) :: volume
      integer(TC) :: k
      k  = tree%mask_out
      if (tree%branch(k)%has_children) then
         call compute_volume_x (tree%branch(k), k, volume, .true.)
      else
         volume = 1
      end if
    contains
      recursive subroutine compute_volume_x (b, k, volume, initial)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        real(default), intent(out) :: volume
        logical, intent(in) :: initial
        integer(TC) :: k1, k2
        real(default) :: v1, v2
        k1 = b%daughter(1);  k2 = b%daughter(2)
        if (tree%branch(k1)%has_children) then
           call compute_volume_x (tree%branch(k1), k1, v1, .false.)
        else
           v1 = 1
        end if
        if (tree%branch(k2)%has_children) then
```

```
          call compute_volume_x (tree%branch(k2), k2, v2, .false.)
      else
         v2 = 1
      end if
      if (initial) then
         volume = v1 * v2 / (4 * twopi5)
      else
         volume = v1 * v2 * sqrts**2 / (4 * twopi2)
      end if
    end subroutine compute_volume_x
  end subroutine phs_tree_compute_volume
```

Determine momenta

This is done in two steps: First the masses are determined. This step may
fail, in which case `ok` is set to false. If successful, we generate angles and
the actual momenta. The array `decay_p` serves for transferring the individual
three-momenta of the daughter particles in their mother rest frame from the
mass generation to the momentum generation step.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_compute_momenta_from_x
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_compute_momenta_from_x &
        (tree, prt, factor, volume, sqrts, x, ok)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(inout) :: prt
      real(default), intent(out) :: factor, volume
      real(default), intent(in) :: sqrts
      real(default), dimension(:), intent(in) :: x
      logical, intent(out) :: ok
      real(default), dimension(tree%mask_out) :: decay_p
      integer :: n1, n2
      n1 = tree%n_msq
      n2 = n1 + tree%n_angles
      call phs_tree_set_msq &
          (tree, prt, factor, volume, decay_p, sqrts, x(1:n1), ok)
      if (ok) call phs_tree_set_angles &
          (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
    end subroutine phs_tree_compute_momenta_from_x
```

Mass generation is done recursively. The `ok` flag causes the filled tree to be
discarded if set to `.false.`. This happens if a three-momentum turns out to be
imaginary, indicating impossible kinematics. The index `ix` tells us how far we
have used up the input array `x`.

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_set_msq &
        (tree, prt, factor, volume, decay_p, sqrts, x, ok)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(inout) :: prt
      real(default), intent(out) :: factor, volume
      real(default), dimension(:), intent(out) :: decay_p
```

```fortran
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(in) :: x
    logical, intent(out) :: ok
    integer :: ix
    integer(TC) :: k
    real(default) :: m_tot
    ok =.true.
    ix = 1
    k  = tree%mask_out
    m_tot = tree%mass_sum(k)
    decay_p(k) = 0.
    if (m_tot < sqrts .or. k == 1) then
       if (tree%branch(k)%has_children) then
          call set_msq_x (tree%branch(k), k, factor, volume, .true.)
       else
          factor = 1
          volume = 1
       end if
    else
       ok = .false.
    end if
  contains
    recursive subroutine set_msq_x (b, k, factor, volume, initial)
      type(phs_branch_t), intent(in) :: b
      integer(TC), intent(in) :: k
      real(default), intent(out) :: factor, volume
      logical, intent(in) :: initial
      real(default) :: msq, m, m_min, m_max, m1, m2, msq1, msq2, lda, rlda
      integer(TC) :: k1, k2
      real(default) :: f1, f2, v1, v2
      k1 = b%daughter(1);   k2 = b%daughter(2)
      if (tree%branch(k1)%has_children) then
         call set_msq_x (tree%branch(k1), k1, f1, v1, .false.)
         if (.not.ok) return
      else
         f1 = 1;   v1 = 1
      end if
      if (tree%branch(k2)%has_children) then
         call set_msq_x (tree%branch(k2), k2, f2, v2, .false.)
         if (.not.ok) return
      else
         f2 = 1;   v2 = 1
      end if
      m_min = tree%mass_sum(k)
      if (initial) then
         msq = sqrts**2
         m = sqrts
         m_max = sqrts
         factor = f1 * f2
         volume = v1 * v2 / (4 * twopi5)
      else
         m_max = sqrts - m_tot + m_min
         call mapping_compute_msq_from_x &
              (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
```

```
               x(ix)); ix = ix + 1
          if (msq >= 0) then
             m = sqrt (msq)
             factor = f1 * f2 * factor
             volume = v1 * v2 * sqrts**2 / (4 * twopi2)
             call phs_prt_set_msq (prt(k), msq)
             call phs_prt_set_defined (prt(k))
          else
             ok = .false.
          end if
       end if
       if (ok) then
          msq1 = phs_prt_get_msq (prt(k1));  m1 = sqrt (msq1)
          msq2 = phs_prt_get_msq (prt(k2));  m2 = sqrt (msq2)
          lda = lambda (msq, msq1, msq2)
          if (lda > 0 .and. m > m1 + m2 .and. m <= m_max) then
             rlda = sqrt (lda)
             decay_p(k1) = rlda / (2*m)
             decay_p(k2) = - decay_p(k1)
             factor = rlda / msq * factor
          else
             ok = .false.
          end if
       end if
     end subroutine set_msq_x
   end subroutine phs_tree_set_msq
```

The heart of phase space generation: Now we have the invariant masses, let us generate angles. At each branch, we take a Lorentz transformation and augment it by a boost to the current particle rest frame, and by rotations ϕ and θ around the z and y axis, respectively. This transformation is passed down to the daughter particles, if present.

⟨PHS trees: procedures⟩+≡
```
   subroutine phs_tree_set_angles (tree, prt, factor, decay_p, sqrts, x)
     type(phs_tree_t), intent(inout) :: tree
     type(phs_prt_t), dimension(:), intent(inout) :: prt
     real(default), intent(inout) :: factor
     real(default), dimension(:), intent(in) :: decay_p
     real(default), intent(in) :: sqrts
     real(default), dimension(:), intent(in) :: x
     integer :: ix
     integer(TC) :: k
     ix = 1
     k  = tree%mask_out
     call set_angles_x (tree%branch(k), k)
   contains
     recursive subroutine set_angles_x (b, k, L0)
       type(phs_branch_t), intent(in) :: b
       integer(TC), intent(in) :: k
       type(lorentz_transformation_t), intent(in), optional :: L0
       real(default) :: m, msq, ct, st, phi, f, E, p, bg
       type(lorentz_transformation_t) :: L, LL
       integer(TC) :: k1, k2
```

1446

```
      type(vector3_t) :: axis
      p = decay_p(k)
      msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
      E = sqrt (msq + p**2)
      if (present (L0)) then
          call phs_prt_set_momentum (prt(k), L0 * vector4_moving (E,p,3))
      else
          call phs_prt_set_momentum (prt(k), vector4_moving (E,p,3))
      end if
      call phs_prt_set_defined (prt(k))
      if (b%has_children) then
          k1 = b%daughter(1)
          k2 = b%daughter(2)
          if (m > 0) then
              bg = p / m
          else
              bg = 0
          end if
          phi = x(ix) * twopi;  ix = ix + 1
          call mapping_compute_ct_from_x &
              (tree%mapping(k), sqrts**2, ct, st, f, x(ix));  ix = ix + 1
          factor = factor * f
          if (.not. b%has_friend) then
              L = LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), bg)
              !!! The function above is equivalent to:
              ! L = boost (bg,3) * rotation (phi,3) * rotation (ct,st,2)
          else
              LL = boost (-bg,3);  if (present (L0))  LL = LL * inverse(L0)
              axis = space_part ( &
                  LL * phs_prt_get_momentum (prt(tree%branch(k)%friend)) )
              L = boost(bg,3) * rotation_to_2nd (vector3_canonical(3), axis) &
                  * LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), 0._default)
          end if
          if (present (L0))  L = L0 * L
          call set_angles_x (tree%branch(k1), k1, L)
          call set_angles_x (tree%branch(k2), k2, L)
      end if
    end subroutine set_angles_x
  end subroutine phs_tree_set_angles
```

Recover random numbers

For the other channels we want to compute the random numbers that would
have generated the momenta that we already know.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_compute_x_from_momenta
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_compute_x_from_momenta (tree, prt, factor, sqrts, x)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(in) :: prt
      real(default), intent(out) :: factor
      real(default), intent(in) :: sqrts
```

```
      real(default), dimension(:), intent(inout) :: x
      real(default), dimension(tree%mask_out) :: decay_p
      integer :: n1, n2
      n1 = tree%n_msq
      n2 = n1 + tree%n_angles
      call phs_tree_get_msq &
           (tree, prt, factor, decay_p, sqrts, x(1:n1))
      call phs_tree_get_angles &
           (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
    end subroutine phs_tree_compute_x_from_momenta
```

The inverse operation follows exactly the same steps. The tree is `inout` because it contains mappings whose parameters can be reset when the mapping is applied.

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_get_msq (tree, prt, factor, decay_p, sqrts, x)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(in) :: prt
      real(default), intent(out) :: factor
      real(default), dimension(:), intent(out) :: decay_p
      real(default), intent(in) :: sqrts
      real(default), dimension(:), intent(inout) :: x
      integer :: ix
      integer(TC) :: k
      real(default) :: m_tot
      ix = 1
      k  = tree%mask_out
      m_tot = tree%mass_sum(k)
      decay_p(k) = 0.
      if (tree%branch(k)%has_children) then
         call get_msq_x (tree%branch(k), k, factor, .true.)
      else
         factor = 1
      end if
    contains
      recursive subroutine get_msq_x (b, k, factor, initial)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        real(default), intent(out) :: factor
        logical, intent(in) :: initial
        real(default) :: msq, m, m_min, m_max, msq1, msq2, lda, rlda
        integer(TC) :: k1, k2
        real(default) :: f1, f2
        k1 = b%daughter(1);   k2 = b%daughter(2)
        if (tree%branch(k1)%has_children) then
           call get_msq_x (tree%branch(k1), k1, f1, .false.)
        else
           f1 = 1
        end if
        if (tree%branch(k2)%has_children) then
           call get_msq_x (tree%branch(k2), k2, f2, .false.)
        else
           f2 = 1
```

1448

```
        end if
        m_min = tree%mass_sum(k)
        m_max = sqrts - m_tot + m_min
        msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
        if (initial) then
           factor = f1 * f2
        else
           call mapping_compute_x_from_msq &
                (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
                 x(ix));  ix = ix + 1
           factor = f1 * f2 * factor
        end if
        msq1 = phs_prt_get_msq (prt(k1))
        msq2 = phs_prt_get_msq (prt(k2))
        lda = lambda (msq, msq1, msq2)
        if (lda > 0) then
           rlda = sqrt (lda)
           decay_p(k1) = rlda / (2 * m)
           decay_p(k2) = - decay_p(k1)
           factor = rlda / msq * factor
        else
           decay_p(k1) = 0
           decay_p(k2) = 0
           factor = 0
        end if
      end subroutine get_msq_x
  end subroutine phs_tree_get_msq
```

This subroutine is the most time-critical part of the whole program. Therefore, we do not exactly parallel the angle generation routine above but make sure that things get evaluated only if they are really needed, at the expense of readability. Particularly important is to have as few multiplications of Lorentz transformations as possible.

⟨*PHS trees: procedures*⟩+≡

```
    subroutine phs_tree_get_angles (tree, prt, factor, decay_p, sqrts, x)
      type(phs_tree_t), intent(inout) :: tree
      type(phs_prt_t), dimension(:), intent(in) :: prt
      real(default), intent(inout) :: factor
      real(default), dimension(:), intent(in) :: decay_p
      real(default), intent(in) :: sqrts
      real(default), dimension(:), intent(out) :: x
      integer :: ix
      integer(TC) :: k
      ix = 1
      k  = tree%mask_out
      if (tree%branch(k)%has_children)  call get_angles_x (tree%branch(k), k)
    contains
      recursive subroutine get_angles_x (b, k, ct0, st0, phi0, L0)
        type(phs_branch_t), intent(in) :: b
        integer(TC), intent(in) :: k
        real(default), intent(in), optional :: ct0, st0, phi0
        type(lorentz_transformation_t), intent(in), optional :: L0
        real(default) :: cp0, sp0, m, msq, ct, st, phi, bg, f
```

1449

```
type(lorentz_transformation_t) :: L, LL
type(vector4_t) :: p1, pf
type(vector3_t) :: n, axis
integer(TC) :: k1, k2, kf
logical :: has_friend, need_L
k1 = b%daughter(1)
k2 = b%daughter(2)
kf = b%friend
has_friend = b%has_friend
if (present(L0)) then
   p1 = L0 * phs_prt_get_momentum (prt(k1))
   if (has_friend)  pf = L0 * phs_prt_get_momentum (prt(kf))
else
   p1 = phs_prt_get_momentum (prt(k1))
   if (has_friend)  pf = phs_prt_get_momentum (prt(kf))
end if
if (present(phi0)) then
   cp0 = cos (phi0)
   sp0 = sin (phi0)
end if
msq = phs_prt_get_msq (prt(k)); m = sqrt (msq)
if (m > 0) then
   bg = decay_p(k) / m
else
   bg = 0
end if
if (has_friend) then
   if (present (phi0)) then
      axis = axis_from_p_r3_r2_b3 (pf, cp0, -sp0, ct0, -st0, -bg)
      LL = rotation_to_2nd (axis, vector3_canonical (3)) &
           * LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
   else
      axis = axis_from_p_b3 (pf, -bg)
      LL = rotation_to_2nd (axis, vector3_canonical(3))
      if (bg /= 0)  LL = LL * boost(-bg, 3)
   end if
   n = space_part (LL * p1)
else if (present (phi0)) then
   n = axis_from_p_r3_r2_b3 (p1, cp0, -sp0, ct0, -st0, -bg)
else
   n = axis_from_p_b3 (p1, -bg)
end if
phi = azimuthal_angle (n)
x(ix) = phi / twopi;  ix = ix + 1
ct = polar_angle_ct (n)
st = sqrt (1 - ct**2)
call mapping_compute_x_from_ct (tree%mapping(k), sqrts**2, ct, f, &
     x(ix));  ix = ix + 1
factor = factor * f
if (tree%branch(k1)%has_children .or. tree%branch(k2)%has_children) then
   need_L = .true.
   if (has_friend) then
      if (present (L0)) then
         L = LL * L0
```

```
                else
                   L = LL
                end if
             else if (present (L0)) then
                L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg) * L0
             else if (present (phi0)) then
                L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
             else if (bg /= 0) then
                L = boost(-bg, 3)
             else
                need_L = .false.
             end if
             if (need_L) then
                if (tree%branch(k1)%has_children) &
                     call get_angles_x (tree%branch(k1), k1, ct, st, phi, L)
                if (tree%branch(k2)%has_children) &
                     call get_angles_x (tree%branch(k2), k2, ct, st, phi, L)
             else
                if (tree%branch(k1)%has_children) &
                     call get_angles_x (tree%branch(k1), k1, ct, st, phi)
                if (tree%branch(k2)%has_children) &
                     call get_angles_x (tree%branch(k2), k2, ct, st, phi)
             end if
          end if
        end subroutine get_angles_x
    end subroutine phs_tree_get_angles
```

Auxiliary stuff

This calculates all momenta that are not yet known by summing up daughter particle momenta. The external particles must be known. Only composite particles not yet known are calculated.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_combine_particles
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_combine_particles (tree, prt)
      type(phs_tree_t), intent(in) :: tree
      type(phs_prt_t), dimension(:), intent(inout) :: prt
      call combine_particles_x (tree%mask_out)
    contains
      recursive subroutine combine_particles_x (k)
        integer(TC), intent(in) :: k
        integer :: k1, k2
        if (tree%branch(k)%has_children) then
           k1 = tree%branch(k)%daughter(1);   k2 = tree%branch(k)%daughter(2)
           call combine_particles_x (k1)
           call combine_particles_x (k2)
           if (.not. prt(k)%defined) then
              call phs_prt_combine (prt(k), prt(k1), prt(k2))
           end if
        end if
      end subroutine combine_particles_x
```

```
      end subroutine phs_tree_combine_particles
```

The previous routine is to be evaluated at runtime. Instead of scanning trees, we can as well set up a multiplication table. This is generated here. Note that the table is `intent(out)`.

⟨*PHS trees: public*⟩+≡
```
    public :: phs_tree_setup_prt_combinations
```

⟨*PHS trees: procedures*⟩+≡
```
    subroutine phs_tree_setup_prt_combinations (tree, comb)
      type(phs_tree_t), intent(in) :: tree
      integer, dimension(:,:), intent(out) :: comb
      comb = 0
      call setup_prt_combinations_x (tree%mask_out)
    contains
      recursive subroutine setup_prt_combinations_x (k)
        integer(TC), intent(in) :: k
        integer, dimension(2) :: kk
        if (tree%branch(k)%has_children) then
           kk = tree%branch(k)%daughter
           call setup_prt_combinations_x (kk(1))
           call setup_prt_combinations_x (kk(2))
           comb(:,k) = kk
        end if
      end subroutine setup_prt_combinations_x
    end subroutine phs_tree_setup_prt_combinations
```

11.6 The phase-space forest

Simply stated, a phase-space forest is a collection of phase-space trees. More precisely, a `phs_forest` object contains all parameterizations of phase space that WHIZARD will use for a single hard process, prepared in the form of `phs_tree` objects. This is suitable for evaluation by the VAMP integration package: each parameterization (tree) is a valid channel in the multi-channel adaptive integration, and each variable in a tree corresponds to an integration dimension, defined by an appropriate mapping of the $(0, 1)$ interval to the allowed range of the integration variable.

The trees are grouped in groves. The trees (integration channels) within a grove share a common weight, assuming that they are related by some approximate symmetry.

Trees/channels that are related by an exact symmetry are connected by an array of equivalences; each equivalence object holds the data that relate one channel to another.

The phase-space setup, i.e., the detailed structure of trees and forest, are read from a file. Therefore, this module also contains the syntax definition and the parser needed for interpreting this file.

⟨`phs_forests.f90`⟩≡
  ⟨*File header*⟩

  `module phs_forests`

  ⟨*Use kinds*⟩
    `use kinds, only: TC !NODEP!`
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    `use limits, only: FMT_12, FMT_19 !NODEP!`
    `use diagnostics !NODEP!`
    `use lorentz !NODEP!`
    `use unit_tests`
    `use permutations`
    `use ifiles`
    `use syntax_rules`
    `use lexers`
    `use parser`
    `use models`
    `use flavors`
    `use interactions`

    `use phs_base`
    `use mappings`
    `use phs_trees`

  ⟨*Standard module head*⟩

  ⟨*PHS forests: public*⟩

  ⟨*PHS forests: types*⟩

  ⟨*PHS forests: interfaces*⟩

⟨*PHS forests: variables*⟩

```
contains
```

⟨*PHS forests: procedures*⟩

⟨*PHS forests: tests*⟩

```
end module phs_forests
```

11.6.1   Phase-space setup parameters

This transparent container holds the parameters that the algorithm needs for phase-space setup, with reasonable defaults.

The threshold mass (for considering a particle as effectively massless) is specified separately for s- and t-channel. The default is to treat W and Z bosons as massive in the s-channel, but as massless in the t-channel. The b-quark is treated always massless, the t-quark always massive.

⟨*PHS forests: public*⟩≡

```
    public :: phs_parameters_t
```

⟨*PHS forests: types*⟩≡

```
    type :: phs_parameters_t
       real(default) :: sqrts = 0
       real(default) :: m_threshold_s = 50._default
       real(default) :: m_threshold_t = 100._default
       integer :: off_shell = 1
       integer :: t_channel = 2
       logical :: keep_nonresonant = .true.
    end type phs_parameters_t
```

Write phase-space parameters to file.

⟨*PHS forests: public*⟩+≡

```
    public :: phs_parameters_write
```

⟨*PHS forests: procedures*⟩≡

```
    subroutine phs_parameters_write (phs_par, unit)
      type(phs_parameters_t), intent(in) :: phs_par
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A," // FMT_19 // ")") "sqrts         = ", phs_par%sqrts
      write (u, "(3x,A," // FMT_19 // ")") "m_threshold_s = ", phs_par%m_threshold_s
      write (u, "(3x,A," // FMT_19 // ")") "m_threshold_t = ", phs_par%m_threshold_t
      write (u, "(3x,A,I0)") "off_shell = ", phs_par%off_shell
      write (u, "(3x,A,I0)") "t_channel = ", phs_par%t_channel
      write (u, "(3x,A,L1)") "keep_nonresonant = ", phs_par%keep_nonresonant
    end subroutine phs_parameters_write
```

Read phase-space parameters from file.

⟨*PHS forests: public*⟩+≡

```
    public :: phs_parameters_read
```

⟨*PHS forests: procedures*⟩+≡

```
    subroutine phs_parameters_read (phs_par, unit)
      type(phs_parameters_t), intent(out) :: phs_par
      integer, intent(in) :: unit
      character(20) :: dummy
      character :: equals
      read (unit, *)  dummy, equals, phs_par%sqrts
      read (unit, *)  dummy, equals, phs_par%m_threshold_s
      read (unit, *)  dummy, equals, phs_par%m_threshold_t
      read (unit, *)  dummy, equals, phs_par%off_shell
      read (unit, *)  dummy, equals, phs_par%t_channel
      read (unit, *)  dummy, equals, phs_par%keep_nonresonant
    end subroutine phs_parameters_read
```

Comparison.

⟨*PHS forests: interfaces*⟩≡

```
    interface operator(==)
       module procedure phs_parameters_eq
    end interface
    interface operator(/=)
       module procedure phs_parameters_ne
    end interface
```

⟨*PHS forests: procedures*⟩+≡

```
    function phs_parameters_eq (phs_par1, phs_par2) result (equal)
      logical :: equal
      type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
      equal = phs_par1%sqrts == phs_par2%sqrts &
           .and. phs_par1%m_threshold_s == phs_par2%m_threshold_s &
           .and. phs_par1%m_threshold_t == phs_par2%m_threshold_t &
           .and. phs_par1%off_shell == phs_par2%off_shell &
           .and. phs_par1%t_channel == phs_par2%t_channel &
           .and.(phs_par1%keep_nonresonant .eqv. phs_par2%keep_nonresonant)
    end function phs_parameters_eq

    function phs_parameters_ne (phs_par1, phs_par2) result (ne)
      logical :: ne
      type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
      ne = phs_par1%sqrts /= phs_par2%sqrts &
           .or. phs_par1%m_threshold_s /= phs_par2%m_threshold_s &
           .or. phs_par1%m_threshold_t /= phs_par2%m_threshold_t &
           .or. phs_par1%off_shell /= phs_par2%off_shell &
           .or. phs_par1%t_channel /= phs_par2%t_channel &
           .or.(phs_par1%keep_nonresonant .neqv. phs_par2%keep_nonresonant)
    end function phs_parameters_ne
```

11.6.2 Equivalences

This type holds information about equivalences between phase-space trees. We make a linked list, where each node contains the two trees which are equivalent and the corresponding permutation of external particles. Two more arrays are

to be filled: The permutation of mass variables and the permutation of angular variables, where the signature indicates a necessary exchange of daughter branches.

⟨*PHS forests: types*⟩+≡

```
type :: equivalence_t
  private
  integer :: left, right
  type(permutation_t) :: perm
  type(permutation_t) :: msq_perm, angle_perm
  logical, dimension(:), allocatable :: angle_sig
  type(equivalence_t), pointer :: next => null ()
end type equivalence_t
```

⟨*PHS forests: types*⟩+≡

```
type :: equivalence_list_t
  private
  integer :: length = 0
  type(equivalence_t), pointer :: first => null ()
  type(equivalence_t), pointer :: last => null ()
end type equivalence_list_t
```

Append an equivalence to the list

⟨*PHS forests: procedures*⟩+≡

```
subroutine equivalence_list_add (eql, left, right, perm)
  type(equivalence_list_t), intent(inout) :: eql
  integer, intent(in) :: left, right
  type(permutation_t), intent(in) :: perm
  type(equivalence_t), pointer :: eq
  allocate (eq)
  eq%left = left
  eq%right = right
  eq%perm = perm
  if (associated (eql%last)) then
     eql%last%next => eq
  else
     eql%first => eq
  end if
  eql%last => eq
  eql%length = eql%length + 1
end subroutine equivalence_list_add
```

Delete the list contents. Has to be pure because it is called from an elemental subroutine.

⟨*PHS forests: procedures*⟩+≡

```
pure subroutine equivalence_list_final (eql)
  type(equivalence_list_t), intent(inout) :: eql
  type(equivalence_t), pointer :: eq
  do while (associated (eql%first))
     eq => eql%first
     eql%first => eql%first%next
     deallocate (eq)
  end do
```

```
      eql%last => null ()
      eql%length = 0
   end subroutine equivalence_list_final
```

Make a deep copy of the equivalence list. This allows for deep copies of groves and forests.

⟨*PHS forests: interfaces*⟩+≡
```
   interface assignment(=)
      module procedure equivalence_list_assign
   end interface
```

⟨*PHS forests: procedures*⟩+≡
```
   subroutine equivalence_list_assign (eql_out, eql_in)
     type(equivalence_list_t), intent(out) :: eql_out
     type(equivalence_list_t), intent(in) :: eql_in
     type(equivalence_t), pointer :: eq, eq_copy
     eq => eql_in%first
     do while (associated (eq))
        allocate (eq_copy)
        eq_copy = eq
        eq_copy%next => null ()
        if (associated (eql_out%first)) then
           eql_out%last%next => eq_copy
        else
           eql_out%first => eq_copy
        end if
        eql_out%last => eq_copy
        eq => eq%next
     end do
   end subroutine equivalence_list_assign
```

The number of list entries

⟨*PHS forests: procedures*⟩+≡
```
   elemental function equivalence_list_length (eql) result (length)
     integer :: length
     type(equivalence_list_t), intent(in) :: eql
     length = eql%length
   end function equivalence_list_length
```

Recursively write the equivalences list

⟨*PHS forests: procedures*⟩+≡
```
   subroutine equivalence_list_write (eql, unit)
     type(equivalence_list_t), intent(in) :: eql
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     if (associated (eql%first)) then
        call equivalence_write_rec (eql%first, u)
     else
        write (u, *) " [empty]"
     end if
   contains
```

```
      recursive subroutine equivalence_write_rec (eq, u)
        type(equivalence_t), intent(in) :: eq
        integer, intent(in) :: u
        integer :: i
        write (u, "(3x,A,1x,I0,1x,I0,2x,A)", advance="no") &
             "Equivalence:", eq%left, eq%right, "Final state permutation:"
        call permutation_write (eq%perm, u)
        write (u, "(1x,12x,1x,A,1x)", advance="no") &
             "      msq permutation:  "
        call permutation_write (eq%msq_perm, u)
        write (u, "(1x,12x,1x,A,1x)", advance="no") &
             "      angle permutation:"
        call permutation_write (eq%angle_perm, u)
        write (u, "(1x,12x,1x,26x)", advance="no")
        do i = 1, size (eq%angle_sig)
           if (eq%angle_sig(i)) then
              write (u, "(1x,A)", advance="no") "+"
           else
              write (u, "(1x,A)", advance="no") "-"
           end if
        end do
        write (u, *)
        if (associated (eq%next))  call equivalence_write_rec (eq%next, u)
      end subroutine equivalence_write_rec
    end subroutine equivalence_list_write
```

11.6.3 Groves

A grove is a group of trees (phase-space channels) that share a common weight
in the integration. Within a grove, channels can be declared equivalent, so they
also share their integration grids (up to symmetries). The grove contains a list
of equivalences. The `tree_count_offset` is the total number of trees of the
preceding groves; when the trees are counted per forest (integration channels),
the offset has to be added to all tree indices.

⟨*PHS forests: types*⟩+≡
```
    type :: phs_grove_t
       private
       integer :: tree_count_offset
       type(phs_tree_t), dimension(:), allocatable :: tree
       type(equivalence_list_t) :: equivalence_list
    end type phs_grove_t
```

Call `phs_tree_init` which is also elemental:

⟨*PHS forests: procedures*⟩+≡
```
    elemental subroutine phs_grove_init &
         (grove, n_trees, n_in, n_out, n_masses, n_angles)
      type(phs_grove_t), intent(inout) :: grove
      integer, intent(in) :: n_trees, n_in, n_out, n_masses, n_angles
      grove%tree_count_offset = 0
      allocate (grove%tree (n_trees))
      call phs_tree_init (grove%tree, n_in, n_out, n_masses, n_angles)
```

```
    end subroutine phs_grove_init
```

The trees do not have pointer components, thus no call to `phs_tree_final`:

⟨*PHS forests: procedures*⟩+≡
```
    elemental subroutine phs_grove_final (grove)
      type(phs_grove_t), intent(inout) :: grove
      deallocate (grove%tree)
      call equivalence_list_final (grove%equivalence_list)
    end subroutine phs_grove_final
```

Deep copy.

⟨*PHS forests: interfaces*⟩+≡
```
    interface assignment(=)
       module procedure phs_grove_assign0
       module procedure phs_grove_assign1
    end interface
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_grove_assign0 (grove_out, grove_in)
      type(phs_grove_t), intent(out) :: grove_out
      type(phs_grove_t), intent(in) :: grove_in
      grove_out%tree_count_offset = grove_in%tree_count_offset
      if (allocated (grove_in%tree)) then
         allocate (grove_out%tree (size (grove_in%tree)))
         grove_out%tree = grove_in%tree
      end if
      grove_out%equivalence_list = grove_in%equivalence_list
    end subroutine phs_grove_assign0

    subroutine phs_grove_assign1 (grove_out, grove_in)
      type(phs_grove_t), dimension(:), intent(out) :: grove_out
      type(phs_grove_t), dimension(:), intent(in) :: grove_in
      integer :: i
      do i = 1, size (grove_in)
         call phs_grove_assign0 (grove_out(i), grove_in(i))
      end do
    end subroutine phs_grove_assign1
```

Get the global (s-channel) mappings. Implemented as a subroutine which returns an array (slice).

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_grove_assign_s_mappings (grove, mapping)
      type(phs_grove_t), intent(in) :: grove
      type(mapping_t), dimension(:), intent(out) :: mapping
      integer :: i
      if (size (mapping) == size (grove%tree)) then
         do i = 1, size (mapping)
            call phs_tree_assign_s_mapping (grove%tree(i), mapping(i))
         end do
      else
         call msg_bug ("phs_grove_assign_s_mappings: array size mismatch")
      end if
```

1459

```
      end subroutine phs_grove_assign_s_mappings
```

11.6.4 The forest type

This is a collection of trees and associated particles. In a given tree, each branch code corresponds to a particle in the `prt` array. Furthermore, we have an array of mass sums which is independent of the decay tree and of the particular event. The mappings directly correspond to the decay trees, and the decay groves collect the trees in classes. The permutation list consists of all permutations of outgoing particles that map the decay forest onto itself.

The particle codes `flv` (one for each external particle) are needed for determining masses and such. The trees and associated information are collected in the `grove` array, together with a lookup table that associates tree indices to groves. Finally, the `prt` array serves as workspace for phase-space evaluation.

The `prt_combination` is a list of index pairs, namely the particle momenta pairs that need to be combined in order to provide all momentum combinations that the phase-space trees need to know.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_t
```

⟨*PHS forests: types*⟩+≡
```
    type :: phs_forest_t
      private
      integer :: n_in, n_out, n_tot
      integer :: n_masses, n_angles, n_dimensions
      integer :: n_trees, n_equivalences
      type(flavor_t), dimension(:), allocatable :: flv
      type(phs_grove_t), dimension(:), allocatable :: grove
      integer, dimension(:), allocatable :: grove_lookup
      type(phs_prt_t), dimension(:), allocatable :: prt_in
      type(phs_prt_t), dimension(:), allocatable :: prt_out
      type(phs_prt_t), dimension(:), allocatable :: prt
      integer(TC), dimension(:,:), allocatable :: prt_combination
      type(mapping_t), dimension(:), allocatable :: s_mapping
    contains
    ⟨PHS forests: phs forest: TBP⟩
    end type phs_forest_t
```

The initialization merely allocates memory. We have to know how many trees there are in each grove, so we can initialize everything. The number of groves is the size of the `n_tree` array.

In the `grove_lookup` table we store the grove index that belongs to each absolute tree index. The difference between the absolute index and the relative (to the grove) index is stored, for each grove, as `tree_count_offset`.

The particle array is allocated according to the total number of branches each tree has, but not filled.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_init
```

⟨PHS forests: procedures⟩+≡

```
subroutine phs_forest_init (forest, n_tree, n_in, n_out)
  type(phs_forest_t), intent(inout) :: forest
  integer, dimension(:), intent(in) :: n_tree
  integer, intent(in) :: n_in, n_out
  integer :: g, count, k_root
  forest%n_in = n_in
  forest%n_out = n_out
  forest%n_tot = n_in + n_out
  forest%n_masses = max (n_out - 2, 0)
  forest%n_angles = max (2*n_out - 2, 0)
  forest%n_dimensions = forest%n_masses + forest%n_angles
  forest%n_trees = sum (n_tree)
  forest%n_equivalences = 0
  allocate (forest%grove (size (n_tree)))
  call phs_grove_init &
       (forest%grove, n_tree, n_in, n_out, forest%n_masses, forest%n_angles)
  allocate (forest%grove_lookup (forest%n_trees))
  count = 0
  do g = 1, size (forest%grove)
     forest%grove(g)%tree_count_offset = count
     forest%grove_lookup (count+1:count+n_tree(g)) = g
     count = count + n_tree(g)
  end do
  allocate (forest%prt_in  (n_in))
  allocate (forest%prt_out (n_out))
  k_root = 2**forest%n_tot - 1
  allocate (forest%prt (k_root))
  allocate (forest%prt_combination (2, k_root))
  allocate (forest%s_mapping (forest%n_trees))
end subroutine phs_forest_init
```

Assign the global (s-channel) mappings.

⟨PHS forests: public⟩+≡

```
public :: phs_forest_set_s_mappings
```

⟨PHS forests: procedures⟩+≡

```
subroutine phs_forest_set_s_mappings (forest)
  type(phs_forest_t), intent(inout) :: forest
  integer :: g, i0, i1, n
  do g = 1, size (forest%grove)
     call phs_forest_get_grove_bounds (forest, g, i0, i1, n)
     call phs_grove_assign_s_mappings &
          (forest%grove(g), forest%s_mapping(i0:i1))
  end do
end subroutine phs_forest_set_s_mappings
```

The grove finalizer is called because it contains the equivalence list:

⟨PHS forests: public⟩+≡

```
public :: phs_forest_final
```

⟨PHS forests: procedures⟩+≡

```
subroutine phs_forest_final (forest)
  type(phs_forest_t), intent(inout) :: forest
```

```
      if (allocated (forest%grove)) then
         call phs_grove_final (forest%grove)
         deallocate (forest%grove)
      end if
      if (allocated (forest%grove_lookup))  deallocate (forest%grove_lookup)
      if (allocated (forest%prt))  deallocate (forest%prt)
      if (allocated (forest%s_mapping))  deallocate (forest%s_mapping)
   end subroutine phs_forest_final
```

11.6.5 Screen output

Write the particles that are non-null, then the trees which point to them:

⟨*PHS forests: public*⟩+≡
```
   public :: phs_forest_write
```

⟨*PHS forests: phs forest: TBP*⟩≡
```
   procedure :: write => phs_forest_write
```

⟨*PHS forests: procedures*⟩+≡
```
   subroutine phs_forest_write (forest, unit)
     class(phs_forest_t), intent(in) :: forest
     integer, intent(in), optional :: unit
     integer :: u
     integer :: i, g, k
     u = output_unit (unit);  if (u < 0)  return
     write (u, "(1x,A)") "Phase space forest:"
     write (u, "(3x,A,I0)") "n_in  = ", forest%n_in
     write (u, "(3x,A,I0)") "n_out = ", forest%n_out
     write (u, "(3x,A,I0)") "n_tot = ", forest%n_tot
     write (u, "(3x,A,I0)") "n_masses = ", forest%n_masses
     write (u, "(3x,A,I0)") "n_angles = ", forest%n_angles
     write (u, "(3x,A,I0)") "n_dim    = ", forest%n_dimensions
     write (u, "(3x,A,I0)") "n_trees  = ", forest%n_trees
     write (u, "(3x,A,I0)") "n_equiv  = ", forest%n_equivalences
     write (u, "(3x,A)", advance="no") "flavors  ="
     if (allocated (forest%flv)) then
        do i = 1, size (forest%flv)
           write (u, "(1x,I0)", advance="no")  flavor_get_pdg (forest%flv(i))
        end do
        write (u, "(A)")
     else
        write (u, "(1x,A)") "[empty]"
     end if
     write (u, "(1x,A)") "Particle combinations:"
     if (allocated (forest%prt_combination)) then
        do k = 1, size (forest%prt_combination, 2)
           if (forest%prt_combination(1, k) /= 0) then
              write (u, "(3x,I0,1x,'<=',1x,I0,1x,'+',1x,I0)") &
                   k, forest%prt_combination(:,k)
           end if
        end do
     else
        write (u, "(3x,A)") "  [empty]"
```

```
      end if
      write (u, "(1x,A)") "Groves and trees:"
      if (allocated (forest%grove)) then
         do g = 1, size (forest%grove)
            write (u, "(3x,A,1x,I0)") "Grove    ", g
            call phs_grove_write (forest%grove(g), unit)
         end do
      else
         write (u, "(3x,A)") "  [empty]"
      end if
      write (u, "(1x,A,I0)") "Total number of equivalences: ", &
           forest%n_equivalences
      write (u, "(A)")
      write (u, "(1x,A)") "Global s-channel mappings:"
      if (allocated (forest%s_mapping)) then
         do i = 1, size (forest%s_mapping)
            associate (mapping => forest%s_mapping(i))
              if (mapping_is_s_channel (mapping) &
                   .or. mapping_is_on_shell (mapping)) then
                 write (u, "(1x,I0,':',1x)", advance="no")  i
                 call mapping_write (forest%s_mapping(i), unit)
              end if
            end associate
         end do
      else
         write (u, "(3x,A)") "  [empty]"
      end if
      write (u, "(A)")
      write (u, "(1x,A)") "Incoming particles:"
      if (allocated (forest%prt_in)) then
         if (any (phs_prt_is_defined (forest%prt_in))) then
            do i = 1, size (forest%prt_in)
               if (phs_prt_is_defined (forest%prt_in(i))) then
                  write (u, "(1x,A,1x,I0)")  "Particle", i
                  call phs_prt_write (forest%prt_in(i), u)
               end if
            end do
         else
            write (u, "(3x,A)")  "[all undefined]"
         end if
      else
         write (u, "(3x,A)")  "  [empty]"
      end if
      write (u, "(A)")
      write (u, "(1x,A)") "Outgoing particles:"
      if (allocated (forest%prt_out)) then
         if (any (phs_prt_is_defined (forest%prt_out))) then
            do i = 1, size (forest%prt_out)
               if (phs_prt_is_defined (forest%prt_out(i))) then
                  write (u, "(1x,A,1x,I0)")  "Particle", i
                  call phs_prt_write (forest%prt_out(i), u)
               end if
            end do
         else
```

```
               write (u, "(3x,A)")  "[all undefined]"
            end if
      else
         write (u, "(1x,A)")  "  [empty]"
      end if
      write (u, "(A)")
      write (u, "(1x,A)") "Tree particles:"
      if (allocated (forest%prt)) then
         if (any (phs_prt_is_defined (forest%prt))) then
            do i = 1, size (forest%prt)
               if (phs_prt_is_defined (forest%prt(i))) then
                  write (u, "(1x,A,1x,I0)")  "Particle", i
                  call phs_prt_write (forest%prt(i), u)
               end if
            end do
         else
            write (u, "(3x,A)")  "[all undefined]"
         end if
      else
         write (u, "(3x,A)")  "  [empty]"
      end if
    end subroutine phs_forest_write

    subroutine phs_grove_write (grove, unit)
      type(phs_grove_t), intent(in) :: grove
      integer, intent(in), optional :: unit
      integer :: u
      integer :: t
      u = output_unit (unit);  if (u < 0)  return
      do t = 1, size (grove%tree)
         write (u, "(3x,A,I0)") "Tree      ", t
         call phs_tree_write (grove%tree(t), unit)
      end do
      write (u, "(1x,A)") "Equivalence list:"
      call equivalence_list_write (grove%equivalence_list, unit)
    end subroutine phs_grove_write
```

Deep copy.

⟨PHS forests: public⟩+≡
```
    public :: assignment(=)
```

⟨PHS forests: interfaces⟩+≡
```
    interface assignment(=)
       module procedure phs_forest_assign
    end interface
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_assign (forest_out, forest_in)
      type(phs_forest_t), intent(out) :: forest_out
      type(phs_forest_t), intent(in) :: forest_in
      forest_out%n_in  = forest_in%n_in
      forest_out%n_out = forest_in%n_out
      forest_out%n_tot = forest_in%n_tot
      forest_out%n_masses = forest_in%n_masses
```

```
      forest_out%n_angles = forest_in%n_angles
      forest_out%n_dimensions  = forest_in%n_dimensions
      forest_out%n_trees  = forest_in%n_trees
      forest_out%n_equivalences  = forest_in%n_equivalences
      if (allocated (forest_in%flv)) then
         allocate (forest_out%flv (size (forest_in%flv)))
         forest_out%flv = forest_in%flv
      end if
      if (allocated (forest_in%grove)) then
         allocate (forest_out%grove (size (forest_in%grove)))
         forest_out%grove = forest_in%grove
      end if
      if (allocated (forest_in%grove_lookup)) then
         allocate (forest_out%grove_lookup (size (forest_in%grove_lookup)))
         forest_out%grove_lookup = forest_in%grove_lookup
      end if
      if (allocated (forest_in%prt_in)) then
         allocate (forest_out%prt_in (size (forest_in%prt_in)))
         forest_out%prt_in = forest_in%prt_in
      end if
      if (allocated (forest_in%prt_out)) then
         allocate (forest_out%prt_out (size (forest_in%prt_out)))
         forest_out%prt_out = forest_in%prt_out
      end if
      if (allocated (forest_in%prt)) then
         allocate (forest_out%prt (size (forest_in%prt)))
         forest_out%prt = forest_in%prt
      end if
      if (allocated (forest_in%s_mapping)) then
         allocate (forest_out%s_mapping (size (forest_in%s_mapping)))
         forest_out%s_mapping = forest_in%s_mapping
      end if
      if (allocated (forest_in%prt_combination)) then
         allocate (forest_out%prt_combination &
              (2, size (forest_in%prt_combination, 2)))
         forest_out%prt_combination = forest_in%prt_combination
      end if
    end subroutine phs_forest_assign
```

11.6.6   Accessing contents

Get the number of integration parameters

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_get_n_parameters
```

⟨*PHS forests: procedures*⟩+≡
```
    function phs_forest_get_n_parameters (forest) result (n)
      integer :: n
      type(phs_forest_t), intent(in) :: forest
      n = forest%n_dimensions
    end function phs_forest_get_n_parameters
```

Get the number of integration channels

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_get_n_channels
```

⟨PHS forests: procedures⟩+≡
```
    function phs_forest_get_n_channels (forest) result (n)
      integer :: n
      type(phs_forest_t), intent(in) :: forest
      n = forest%n_trees
    end function phs_forest_get_n_channels
```

Get the number of groves

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_get_n_groves
```

⟨PHS forests: procedures⟩+≡
```
    function phs_forest_get_n_groves (forest) result (n)
      integer :: n
      type(phs_forest_t), intent(in) :: forest
      n = size (forest%grove)
    end function phs_forest_get_n_groves
```

Get the index bounds for a specific grove.

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_get_grove_bounds
```

⟨PHS forests: procedures⟩+≡
```
    subroutine phs_forest_get_grove_bounds (forest, g, i0, i1, n)
      type(phs_forest_t), intent(in) :: forest
      integer, intent(in) :: g
      integer, intent(out) :: i0, i1, n
      n = size (forest%grove(g)%tree)
      i0 = forest%grove(g)%tree_count_offset + 1
      i1 = forest%grove(g)%tree_count_offset + n
    end subroutine phs_forest_get_grove_bounds
```

Get the number of equivalences

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_get_n_equivalences
```

⟨PHS forests: procedures⟩+≡
```
    function phs_forest_get_n_equivalences (forest) result (n)
      integer :: n
      type(phs_forest_t), intent(in) :: forest
      n = forest%n_equivalences
    end function phs_forest_get_n_equivalences
```

Return true if a particular channel has a global (s-channel) mapping; also return
the resonance mass and width for this mapping.

⟨PHS forests: public⟩+≡
```
    public :: phs_forest_get_s_mapping
    public :: phs_forest_get_on_shell
```

1466

⟨PHS forests: procedures⟩+≡

```
subroutine phs_forest_get_s_mapping (forest, channel, flag, mass, width)
  type(phs_forest_t), intent(in) :: forest
  integer, intent(in) :: channel
  logical, intent(out) :: flag
  real(default), intent(out) :: mass, width
  flag = mapping_is_s_channel (forest%s_mapping(channel))
  if (flag) then
     mass = mapping_get_mass (forest%s_mapping(channel))
     width = mapping_get_width (forest%s_mapping(channel))
  else
     mass = 0
     width = 0
  end if
end subroutine phs_forest_get_s_mapping

subroutine phs_forest_get_on_shell (forest, channel, flag, mass)
  type(phs_forest_t), intent(in) :: forest
  integer, intent(in) :: channel
  logical, intent(out) :: flag
  real(default), intent(out) :: mass
  flag = mapping_is_on_shell (forest%s_mapping(channel))
  if (flag) then
     mass = mapping_get_mass (forest%s_mapping(channel))
  else
     mass = 0
  end if
end subroutine phs_forest_get_on_shell
```

11.6.7 Read the phase space setup from file

The phase space setup is stored in a file. The file may be generated by the
cascades module below, or by other means. This file has to be read and parsed
to create the PHS forest as the internal phase-space representation.

Create lexer and syntax:

⟨PHS forests: procedures⟩+≡

```
subroutine define_phs_forest_syntax (ifile)
  type(ifile_t) :: ifile
  call ifile_append (ifile, "SEQ phase_space_list = process_phase_space*")
  call ifile_append (ifile, "SEQ process_phase_space = " &
       // "process_def process_header phase_space")
  call ifile_append (ifile, "SEQ process_def = process process_list")
  call ifile_append (ifile, "KEY process")
  call ifile_append (ifile, "LIS process_list = process_tag*")
  call ifile_append (ifile, "IDE process_tag")
  call ifile_append (ifile, "SEQ process_header = " &
       // "md5sum_process = md5sum " &
       // "md5sum_model_par = md5sum " &
       // "md5sum_phs_config = md5sum " &
       // "sqrts = real " &
       // "m_threshold_s = real " &
       // "m_threshold_t = real " &
```

```
          // "off_shell = integer " &
          // "t_channel = integer " &
          // "keep_nonresonant = logical")
    call ifile_append (ifile, "KEY '='")
    call ifile_append (ifile, "KEY md5sum_process")
    call ifile_append (ifile, "KEY md5sum_model_par")
    call ifile_append (ifile, "KEY md5sum_phs_config")
    call ifile_append (ifile, "KEY sqrts")
    call ifile_append (ifile, "KEY m_threshold_s")
    call ifile_append (ifile, "KEY m_threshold_t")
    call ifile_append (ifile, "KEY off_shell")
    call ifile_append (ifile, "KEY t_channel")
    call ifile_append (ifile, "KEY keep_nonresonant")
    call ifile_append (ifile, "QUO md5sum = '""' ... '""'")
    call ifile_append (ifile, "REA real")
    call ifile_append (ifile, "INT integer")
    call ifile_append (ifile, "IDE logical")
    call ifile_append (ifile, "SEQ phase_space = grove_def+")
    call ifile_append (ifile, "SEQ grove_def = grove tree_def+")
    call ifile_append (ifile, "KEY grove")
    call ifile_append (ifile, "SEQ tree_def = tree bincodes mapping*")
    call ifile_append (ifile, "KEY tree")
    call ifile_append (ifile, "SEQ bincodes = bincode*")
    call ifile_append (ifile, "INT bincode")
    call ifile_append (ifile, "SEQ mapping = map bincode channel pdg")
    call ifile_append (ifile, "KEY map")
    call ifile_append (ifile, "ALT channel = &
          &s_channel | t_channel | u_channel | &
          &collinear | infrared | radiation | on_shell")
    call ifile_append (ifile, "KEY s_channel")
    ! call ifile_append (ifile, "KEY t_channel")    !!! Key already exists
    call ifile_append (ifile, "KEY u_channel")
    call ifile_append (ifile, "KEY collinear")
    call ifile_append (ifile, "KEY infrared")
    call ifile_append (ifile, "KEY radiation")
    call ifile_append (ifile, "KEY on_shell")
    call ifile_append (ifile, "INT pdg")
  end subroutine define_phs_forest_syntax
```

The model-file syntax and lexer are fixed, therefore stored as module variables:

⟨*PHS forests: variables*⟩≡
```
    type(syntax_t), target, save :: syntax_phs_forest
```

⟨*PHS forests: public*⟩+≡
```
    public :: syntax_phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine syntax_phs_forest_init ()
      type(ifile_t) :: ifile
      call define_phs_forest_syntax (ifile)
      call syntax_init (syntax_phs_forest, ifile)
      call ifile_final (ifile)
    end subroutine syntax_phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine lexer_init_phs_forest (lexer)
      type(lexer_t), intent(out) :: lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "", &
           special_class = ["="] , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_phs_forest))
    end subroutine lexer_init_phs_forest
```

⟨*PHS forests: public*⟩+≡
```
    public :: syntax_phs_forest_final
```
⟨*PHS forests: procedures*⟩+≡
```
    subroutine syntax_phs_forest_final ()
      call syntax_final (syntax_phs_forest)
    end subroutine syntax_phs_forest_final
```

⟨*PHS forests: public*⟩+≡
```
    public :: syntax_phs_forest_write
```
⟨*PHS forests: procedures*⟩+≡
```
    subroutine syntax_phs_forest_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_phs_forest, unit)
    end subroutine syntax_phs_forest_write
```

The concrete parser and interpreter. Generate an input stream for the external `unit`, read the parse tree (with given `syntax` and `lexer`) from this stream, and transfer the contents of the parse tree to the PHS `forest`.

We look for the matching `process` tag, count groves and trees for initializing the `forest`, and fill the trees.

If the optional parameters are set, compare the parameters stored in the file to those. Set `match` true if everything agrees.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_read
```
⟨*PHS forests: interfaces*⟩+≡
```
    interface phs_forest_read
       module procedure phs_forest_read_file
       module procedure phs_forest_read_unit
       module procedure phs_forest_read_parse_tree
    end interface
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_read_file &
         (forest, filename, process_id, n_in, n_out, model, found, &
          md5sum_process, md5sum_model_par, &
          md5sum_phs_config, phs_par, match)
      type(phs_forest_t), intent(out) :: forest
      type(string_t), intent(in) :: filename
```

```
    type(string_t), intent(in) :: process_id
    integer, intent(in) :: n_in, n_out
    type(model_t), intent(in), target :: model
    logical, intent(out) :: found
    character(32), intent(in), optional :: &
         md5sum_process, md5sum_model_par, md5sum_phs_config
    type(phs_parameters_t), intent(in), optional :: phs_par
    logical, intent(out), optional :: match
    type(parse_tree_t), target :: parse_tree
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_phs_forest (lexer)
    call stream_init (stream, char (filename))
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
    call phs_forest_read (forest, parse_tree, &
         process_id, n_in, n_out, model, found, &
         md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    call stream_final (stream)
    call lexer_final (lexer)
    call parse_tree_final (parse_tree)
  end subroutine phs_forest_read_file

  subroutine phs_forest_read_unit &
       (forest, unit, process_id, n_in, n_out, model, found, &
        md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    type(phs_forest_t), intent(out) :: forest
    integer, intent(in) :: unit
    type(string_t), intent(in) :: process_id
    integer, intent(in) :: n_in, n_out
    type(model_t), intent(in), target :: model
    logical, intent(out) :: found
    character(32), intent(in), optional :: &
         md5sum_process, md5sum_model_par, md5sum_phs_config
    type(phs_parameters_t), intent(in), optional :: phs_par
    logical, intent(out), optional :: match
    type(parse_tree_t), target :: parse_tree
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    call lexer_init_phs_forest (lexer)
    call stream_init (stream, unit)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
    call phs_forest_read (forest, parse_tree, &
         process_id, n_in, n_out, model, found, &
         md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
    call stream_final (stream)
    call lexer_final (lexer)
    call parse_tree_final (parse_tree)
  end subroutine phs_forest_read_unit

  subroutine phs_forest_read_parse_tree &
       (forest, parse_tree, process_id, n_in, n_out, model, found, &
        md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
```

```
type(phs_forest_t), intent(out) :: forest
type(parse_tree_t), intent(in), target :: parse_tree
type(string_t), intent(in) :: process_id
integer, intent(in) :: n_in, n_out
type(model_t), intent(in), target :: model
logical, intent(out) :: found
character(32), intent(in), optional :: &
     md5sum_process, md5sum_model_par, md5sum_phs_config
type(phs_parameters_t), intent(in), optional :: phs_par
logical, intent(out), optional :: match
type(parse_node_t), pointer :: node_header, node_phs, node_grove
integer :: n_grove, g
integer, dimension(:), allocatable :: n_tree
integer :: t
! call parse_tree_write (parse_tree)      !!! Debugging
node_header => parse_tree_get_process_ptr (parse_tree, process_id)
found = associated (node_header);  if (.not. found)  return
if (present (match)) then
   call phs_forest_check_input (node_header, &
        md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
   if (.not. match)  return
end if
node_phs => parse_node_get_next_ptr (node_header)
n_grove = parse_node_get_n_sub (node_phs)
allocate (n_tree (n_grove))
do g = 1, n_grove
   node_grove => parse_node_get_sub_ptr (node_phs, g)
   n_tree(g) = parse_node_get_n_sub (node_grove) - 1
end do
call phs_forest_init (forest, n_tree, n_in, n_out)
do g = 1, n_grove
   node_grove => parse_node_get_sub_ptr (node_phs, g)
   do t = 1, n_tree(g)
      call phs_tree_set (forest%grove(g)%tree(t), &
           parse_node_get_sub_ptr (node_grove, t+1), model)
   end do
end do
end subroutine phs_forest_read_parse_tree
```

Check the input for consistency. If any MD5 sum or phase-space parameter
disagrees, the phase-space file cannot be used. The MD5 sum checks are skipped
if the stored MD5 sum is empty.

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_check_input (pn_header, &
     md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
type(parse_node_t), intent(in), target :: pn_header
character(32), intent(in) :: &
     md5sum_process, md5sum_model_par, md5sum_phs_config
type(phs_parameters_t), intent(in), optional :: phs_par
logical, intent(out) :: match
type(parse_node_t), pointer :: pn_md5sum, pn_rval, pn_ival, pn_lval
character(32) :: md5sum
type(phs_parameters_t) :: phs_par_old
```

```
      character(1) :: lstr
      pn_md5sum => parse_node_get_sub_ptr (pn_header, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_process) then
         call msg_message ("Phase space: discarding old configuration &
              &(process changed)")
         match = .false.;  return
      end if
      pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_model_par) then
         call msg_message ("Phase space: discarding old configuration &
              &(model parameters changed)")
         match = .false.;  return
      end if
      pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_phs_config) then
         call msg_message ("Phase space: discarding old configuration &
              &(configuration parameters changed)")
         match = .false.;  return
      end if
      if (present (phs_par)) then
         pn_rval => parse_node_get_next_ptr (pn_md5sum, 3)
         phs_par_old%sqrts = parse_node_get_real (pn_rval)
         pn_rval => parse_node_get_next_ptr (pn_rval, 3)
         phs_par_old%m_threshold_s = parse_node_get_real (pn_rval)
         pn_rval => parse_node_get_next_ptr (pn_rval, 3)
         phs_par_old%m_threshold_t = parse_node_get_real (pn_rval)
         pn_ival => parse_node_get_next_ptr (pn_rval, 3)
         phs_par_old%off_shell = parse_node_get_integer (pn_ival)
         pn_ival => parse_node_get_next_ptr (pn_ival, 3)
         phs_par_old%t_channel = parse_node_get_integer (pn_ival)
         pn_lval => parse_node_get_next_ptr (pn_ival, 3)
         lstr = parse_node_get_string (pn_lval)
         read (lstr, "(L1)")  phs_par_old%keep_nonresonant
         if (phs_par_old /= phs_par) then
            call msg_message &
                 ("Phase space: discarding old configuration &
                 &(configuration parameters changed)")
            match = .false.;  return
         end if
      end if
      match = .true.
   end subroutine phs_forest_check_input
```

Initialize a specific tree in the forest, using the contents of the 'tree' node. First, count the bincodes, allocate an array and read them in, and make the tree. Each t-channel tree is flipped to s-channel. Then, find mappings and initialize them.

⟨*PHS forests: procedures*⟩+≡

```
   subroutine phs_tree_set (tree, node, model)
     type(phs_tree_t), intent(inout) :: tree
     type(parse_node_t), intent(in), target :: node
```

```
      type(model_t), intent(in), target :: model
      type(parse_node_t), pointer :: node_bincodes, node_mapping
      integer :: n_bincodes, offset
      integer(TC), dimension(:), allocatable :: bincode
      integer :: b, n_mappings, m
      integer(TC) :: k
      type(string_t) :: type
      integer :: pdg
      node_bincodes => parse_node_get_sub_ptr (node, 2)
      if (associated (node_bincodes)) then
         select case (char (parse_node_get_rule_key (node_bincodes)))
         case ("bincodes")
            n_bincodes = parse_node_get_n_sub (node_bincodes)
            offset = 2
         case default
            n_bincodes = 0
            offset = 1
         end select
      else
         n_bincodes = 0
         offset = 2
      end if
      allocate (bincode (n_bincodes))
      do b = 1, n_bincodes
         bincode(b) = parse_node_get_integer &
              (parse_node_get_sub_ptr (node_bincodes, b))
      end do
      call phs_tree_from_array (tree, bincode)
      call phs_tree_flip_t_to_s_channel (tree)
      call phs_tree_canonicalize (tree)
      n_mappings = parse_node_get_n_sub (node) - offset
      do m = 1, n_mappings
         node_mapping => parse_node_get_sub_ptr (node, m + offset)
         k = parse_node_get_integer &
              (parse_node_get_sub_ptr (node_mapping, 2))
         type = parse_node_get_key &
              (parse_node_get_sub_ptr (node_mapping, 3))
         pdg = parse_node_get_integer &
              (parse_node_get_sub_ptr (node_mapping, 4))
         call phs_tree_init_mapping (tree, k, type, pdg, model)
      end do
    end subroutine phs_tree_set
```

11.6.8 Preparation

The trees that we read from file do not carry flavor information. This is set
separately:

   The flavor list must be unique for a unique set of masses; if a given particle
can have different flavor, the mass must be degenerate, so we can choose one of
the possible flavor combinations.

⟨*PHS forests: public*⟩+≡

```
    public :: phs_forest_set_flavors
```

⟨*PHS forests: procedures*⟩+≡
```
  subroutine phs_forest_set_flavors (forest, flv)
    type(phs_forest_t), intent(inout) :: forest
    type(flavor_t), dimension(:), intent(in) :: flv
    allocate (forest%flv (size (flv)))
    forest%flv = flv
  end subroutine phs_forest_set_flavors
```

Once the parameter set is fixed, the masses and the widths of the particles
are known and the `mass_sum` arrays as well as the mapping parameters can be
computed. Note that order is important: we first compute the mass sums, then
the ordinary mappings. The resonances obtained here determine the effective
masses, which in turn are used to implement step mappings for resonance decay
products that are not mapped otherwise.

⟨*PHS forests: public*⟩+≡
```
  public :: phs_forest_set_parameters
```

⟨*PHS forests: procedures*⟩+≡
```
  subroutine phs_forest_set_parameters &
      (forest, mapping_defaults, variable_limits)
    type(phs_forest_t), intent(inout) :: forest
    type(mapping_defaults_t), intent(in) :: mapping_defaults
    logical, intent(in) :: variable_limits
    integer :: g, t
    do g = 1, size (forest%grove)
       do t = 1, size (forest%grove(g)%tree)
          call phs_tree_set_mass_sum &
              (forest%grove(g)%tree(t), forest%flv(forest%n_in+1:))
          call phs_tree_set_mapping_parameters (forest%grove(g)%tree(t), &
              mapping_defaults, variable_limits)
          call phs_tree_set_effective_masses (forest%grove(g)%tree(t))
          if (mapping_defaults%step_mapping) then
             call phs_tree_set_step_mappings (forest%grove(g)%tree(t), &
                 mapping_defaults%step_mapping_exp, variable_limits)
          end if
       end do
    end do
  end subroutine phs_forest_set_parameters
```

Generate the particle combination table. Scan all trees and merge their individ-
ual combination tables. At the end, valid entries are non-zero, and they indicate
the indices of a pair of particles to be combined to a new particle. If a particle
is accessible by more than one tree (this is usual), only keep the first possibility.

⟨*PHS forests: public*⟩+≡
```
  public :: phs_forest_setup_prt_combinations
```

⟨*PHS forests: procedures*⟩+≡
```
  subroutine phs_forest_setup_prt_combinations (forest)
    type(phs_forest_t), intent(inout) :: forest
    integer :: g, t
    integer, dimension(:,:), allocatable :: tree_prt_combination
    forest%prt_combination = 0
    allocate (tree_prt_combination (2, size (forest%prt_combination, 2)))
```

```
      do g = 1, size (forest%grove)
         do t = 1, size (forest%grove(g)%tree)
            call phs_tree_setup_prt_combinations &
                 (forest%grove(g)%tree(t), tree_prt_combination)
            where (tree_prt_combination /= 0 .and. forest%prt_combination == 0)
               forest%prt_combination = tree_prt_combination
            end where
         end do
      end do
    end subroutine phs_forest_setup_prt_combinations
```

11.6.9 Accessing the particle arrays

Set the incoming particles from the contents of an interaction.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_set_prt_in
```

⟨*PHS forests: interfaces*⟩+≡
```
    interface phs_forest_set_prt_in
       module procedure phs_forest_set_prt_in_int, phs_forest_set_prt_in_mom
    end interface phs_forest_set_prt_in
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_set_prt_in_int (forest, int, lt_cm_to_lab)
      type(phs_forest_t), intent(inout) :: forest
      type(interaction_t), intent(in) :: int
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call phs_prt_set_momentum (forest%prt_in, &
              inverse (lt_cm_to_lab) * &
              interaction_get_momenta (int, outgoing=.false.))
      else
         call phs_prt_set_momentum (forest%prt_in, &
              interaction_get_momenta (int, outgoing=.false.))
      end if
      call phs_prt_set_msq (forest%prt_in, &
           flavor_get_mass (forest%flv(:forest%n_in)) ** 2)
      call phs_prt_set_defined (forest%prt_in)
    end subroutine phs_forest_set_prt_in_int

    subroutine phs_forest_set_prt_in_mom (forest, mom, lt_cm_to_lab)
      type(phs_forest_t), intent(inout) :: forest
      type(vector4_t), dimension(size (forest%prt_in)), intent(in) :: mom
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call phs_prt_set_momentum (forest%prt_in, &
              inverse (lt_cm_to_lab) * mom)
      else
         call phs_prt_set_momentum (forest%prt_in, mom)
      end if
      call phs_prt_set_msq (forest%prt_in, &
           flavor_get_mass (forest%flv(:forest%n_in)) ** 2)
      call phs_prt_set_defined (forest%prt_in)
```

```
      end subroutine phs_forest_set_prt_in_mom
```

Set the outgoing particles from the contents of an interaction.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_set_prt_out
```

⟨*PHS forests: interfaces*⟩+≡
```
    interface phs_forest_set_prt_out
       module procedure phs_forest_set_prt_out_int, phs_forest_set_prt_out_mom
    end interface phs_forest_set_prt_out
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_set_prt_out_int (forest, int, lt_cm_to_lab)
      type(phs_forest_t), intent(inout) :: forest
      type(interaction_t), intent(in) :: int
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call phs_prt_set_momentum (forest%prt_out, &
              inverse (lt_cm_to_lab) * &
              interaction_get_momenta (int, outgoing=.true.))
      else
         call phs_prt_set_momentum (forest%prt_out, &
              interaction_get_momenta (int, outgoing=.true.))
      end if
      call phs_prt_set_msq (forest%prt_out, &
           flavor_get_mass (forest%flv(forest%n_in+1:)) ** 2)
      call phs_prt_set_defined (forest%prt_out)
    end subroutine phs_forest_set_prt_out_int

    subroutine phs_forest_set_prt_out_mom (forest, mom, lt_cm_to_lab)
      type(phs_forest_t), intent(inout) :: forest
      type(vector4_t), dimension(size (forest%prt_out)), intent(in) :: mom
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call phs_prt_set_momentum (forest%prt_out, &
              inverse (lt_cm_to_lab) * mom)
      else
         call phs_prt_set_momentum (forest%prt_out, mom)
      end if
      call phs_prt_set_msq (forest%prt_out, &
           flavor_get_mass (forest%flv(forest%n_in+1:)) ** 2)
      call phs_prt_set_defined (forest%prt_out)
    end subroutine phs_forest_set_prt_out_mom
```

Combine particles as described by the particle combination table. Particle momentum sums will be calculated only if the resulting particle is contained in at least one of the trees in the current forest. The others are kept undefined.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_combine_particles
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_combine_particles (forest)
      type(phs_forest_t), intent(inout) :: forest
      integer :: k
```

```
      integer, dimension(2) :: kk
      do k = 1, size (forest%prt_combination, 2)
         kk = forest%prt_combination(:,k)
         if (kk(1) /= 0) then
            call phs_prt_combine (forest%prt(k), &
                 forest%prt(kk(1)), forest%prt(kk(2)))
         end if
      end do
    end subroutine phs_forest_combine_particles
```

Extract the outgoing particles and insert into an interaction.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_get_prt_out
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_get_prt_out (forest, int, lt_cm_to_lab)
      type(phs_forest_t), intent(in) :: forest
      type(interaction_t), intent(inout) :: int
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      if (present (lt_cm_to_lab)) then
         call interaction_set_momenta (int, &
              lt_cm_to_lab * &
              phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
      else
         call interaction_set_momenta (int, &
              phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
      end if
    end subroutine phs_forest_get_prt_out
```

Extract the outgoing particle momenta

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_get_momenta_out
```

⟨*PHS forests: procedures*⟩+≡
```
    function phs_forest_get_momenta_out (forest, lt_cm_to_lab) result (p)
      type(phs_forest_t), intent(in) :: forest
      type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
      type(vector4_t), dimension(size (forest%prt_out)) :: p
      p = phs_prt_get_momentum (forest%prt_out)
      if (present (lt_cm_to_lab)) p = p * lt_cm_to_lab
    end function phs_forest_get_momenta_out
```

11.6.10 Find equivalences among phase-space trees

Scan phase space for equivalences. We generate the complete set of unique permutations for the given list of outgoing particles, and use this for scanning equivalences within each grove. We scan all pairs of trees, using all permutations. This implies that trivial equivalences are included, and equivalences between different trees are recorded twice. This is intentional.

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_grove_set_equivalences (grove, perm_array)
```

```
      type(phs_grove_t), intent(inout) :: grove
      type(permutation_t), dimension(:), intent(in) :: perm_array
      type(equivalence_t), pointer :: eq
      integer :: t1, t2, i
      do t1 = 1, size (grove%tree)
         do t2 = 1, size (grove%tree)
            SCAN_PERM: do i = 1, size (perm_array)
               if (phs_tree_equivalent &
                   (grove%tree(t1), grove%tree(t2), perm_array(i))) then
                  call equivalence_list_add &
                       (grove%equivalence_list, t1, t2, perm_array(i))
                  eq => grove%equivalence_list%last
                  call phs_tree_find_msq_permutation &
                       (grove%tree(t1), grove%tree(t2), eq%perm, &
                        eq%msq_perm)
                  call phs_tree_find_angle_permutation &
                       (grove%tree(t1), grove%tree(t2), eq%perm, &
                        eq%angle_perm, eq%angle_sig)
               end if
            end do SCAN_PERM
         end do
      end do
   end subroutine phs_grove_set_equivalences
```

⟨*PHS forests: public*⟩+≡

```
   public :: phs_forest_set_equivalences
```

⟨*PHS forests: procedures*⟩+≡

```
   subroutine phs_forest_set_equivalences (forest)
     type(phs_forest_t), intent(inout) :: forest
     type(permutation_t), dimension(:), allocatable :: perm_array
     integer :: i
     call permutation_array_make &
          (perm_array, flavor_get_pdg (forest%flv(forest%n_in+1:)))
     do i = 1, size (forest%grove)
        call phs_grove_set_equivalences (forest%grove(i), perm_array)
     end do
     forest%n_equivalences = sum (forest%grove%equivalence_list%length)
   end subroutine phs_forest_set_equivalences
```

11.6.11   Interface for channel equivalences

Here, we store the equivalence list in the appropriate containers that the `phs_base`
module provides. There is one separate list for each channel.

⟨*PHS forests: public*⟩+≡

```
   public :: phs_forest_get_equivalences
```

⟨*PHS forests: procedures*⟩+≡

```
   subroutine phs_forest_get_equivalences (forest, channel, azimuthal_dependence)
     type(phs_forest_t), intent(in) :: forest
     type(phs_channel_t), dimension(:), intent(out) :: channel
     logical, intent(in) :: azimuthal_dependence
     integer :: n_masses, n_angles
```

```fortran
integer :: mode_azimuthal_angle
integer, dimension(:), allocatable :: n_eq
type(equivalence_t), pointer :: eq
integer, dimension(:), allocatable :: perm, mode
integer :: g, c, j, left, right
n_masses = forest%n_masses
n_angles = forest%n_angles
allocate (n_eq (forest%n_trees), source = 0)
allocate (perm (forest%n_dimensions))
allocate (mode (forest%n_dimensions), source = EQ_IDENTITY)
do g = 1, size (forest%grove)
   eq => forest%grove(g)%equivalence_list%first
   do while (associated (eq))
      left = eq%left + forest%grove(g)%tree_count_offset
      n_eq(left) = n_eq(left) + 1
      eq => eq%next
   end do
end do
do c = 1, size (channel)
   allocate (channel(c)%eq (n_eq(c)))
   do j = 1, n_eq(c)
      call channel(c)%eq(j)%init (forest%n_dimensions)
   end do
end do
n_eq = 0
if (azimuthal_dependence) then
   mode_azimuthal_angle = EQ_IDENTITY
else
   mode_azimuthal_angle = EQ_INVARIANT
end if
do g = 1, size (forest%grove)
   eq => forest%grove(g)%equivalence_list%first
   do while (associated (eq))
      left = eq%left + forest%grove(g)%tree_count_offset
      right = eq%right + forest%grove(g)%tree_count_offset
      do j = 1, n_masses
         perm(j) = permute (j, eq%msq_perm)
         mode(j) = EQ_IDENTITY
      end do
      do j = 1, n_angles
         perm(n_masses+j) = n_masses + permute (j, eq%angle_perm)
         if (j == 1) then
            mode(n_masses+j) = mode_azimuthal_angle   ! first az. angle
         else if (mod(j,2) == 1) then
            mode(n_masses+j) = EQ_SYMMETRIC           ! other az. angles
         else if (eq%angle_sig(j)) then
            mode(n_masses+j) = EQ_IDENTITY            ! polar angle +
         else
            mode(n_masses+j) = EQ_INVERT              ! polar angle -
         end if
      end do
      n_eq(left) = n_eq(left) + 1
      associate (eq_cur => channel(left)%eq(n_eq(left)))
        eq_cur%c = right
```

1479

```
              eq_cur%perm = perm
              eq_cur%mode = mode
           end associate
           eq => eq%next
        end do
     end do
  end subroutine phs_forest_get_equivalences
```

11.6.12   Phase-space evaluation

Given one row of the x parameter array and the corresponding channel index, compute first all relevant momenta and then recover the remainder of the x array, the Jacobians phs_factor, and the phase-space volume.

The output argument ok indicates whether this was successful.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_evaluate_selected_channel
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_evaluate_selected_channel &
         (forest, channel, active, sqrts, x, phs_factor, volume, ok)
      type(phs_forest_t), intent(inout) :: forest
      integer, intent(in) :: channel
      logical, dimension(:), intent(in) :: active
      real(default), intent(in) :: sqrts
      real(default), dimension(:,:), intent(inout) :: x
      real(default), dimension(:), intent(out) :: phs_factor
      real(default), intent(out) :: volume
      logical, intent(out) :: ok
      integer :: g, t
      integer(TC) :: k, k_root, k_in

      g = forest%grove_lookup (channel)
      t = channel - forest%grove(g)%tree_count_offset
      call phs_prt_set_undefined (forest%prt)
      call phs_prt_set_undefined (forest%prt_out)
      k_in = forest%n_tot

      do k = 1,forest%n_in
         forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
      end do

      do k = 1, forest%n_out
         call phs_prt_set_msq (forest%prt(ibset(0,k-1)), &
              flavor_get_mass (forest%flv(forest%n_in+k)) ** 2)
      end do


      k_root = 2**forest%n_out - 1
      select case (forest%n_in)
      case (1)
         forest%prt(k_root) = forest%prt_in(1)
      case (2)
         call phs_prt_combine &
```

```
            (forest%prt(k_root), forest%prt_in(1), forest%prt_in(2))
       end select
       call phs_tree_compute_momenta_from_x (forest%grove(g)%tree(t), &
            forest%prt, phs_factor(channel), volume, sqrts, x(:,channel), ok)
       if (ok) then
          do k = 1, forest%n_out
             forest%prt_out(k) = forest%prt(ibset(0,k-1))
          end do
       end if
    end subroutine phs_forest_evaluate_selected_channel
```

The remainder: recover x values for all channels except for the current channel.

NOTE: OpenMP not used for the first loop. `combine_particles` is not a channel-local operation.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_evaluate_other_channels
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_evaluate_other_channels &
         (forest, channel, active, sqrts, x, phs_factor, combine)
      type(phs_forest_t), intent(inout) :: forest
      integer, intent(in) :: channel
      logical, dimension(:), intent(in) :: active
      real(default), intent(in) :: sqrts
      real(default), dimension(:,:), intent(inout) :: x
      real(default), dimension(:), intent(inout) :: phs_factor
      logical, intent(in) :: combine
      integer :: g, t, ch, n_channel

      n_channel = forest%n_trees
      if (combine) then
         do ch = 1, n_channel
            if (ch == channel)  cycle
            if (active(ch)) then
               g = forest%grove_lookup(ch)
               t = ch - forest%grove(g)%tree_count_offset
               call phs_tree_combine_particles &
                    (forest%grove(g)%tree(t), forest%prt)
            end if
         end do
      end if

      !OMP PARALLEL PRIVATE (g,t,ch) SHARED(active,forest,sqrts,x,channel)
      !OMP DO SCHEDULE(STATIC)
      do ch = 1, n_channel
         if (ch == channel)  cycle
         if (active(ch)) then
            g = forest%grove_lookup(ch)
            t = ch - forest%grove(g)%tree_count_offset
            call phs_tree_compute_x_from_momenta &
                 (forest%grove(g)%tree(t), &
                 forest%prt, phs_factor(ch), sqrts, x(:,ch))
         end if
      end do
```

```
        !OMP END DO
        !OMP END PARALLEL

        end subroutine phs_forest_evaluate_other_channels
```

The complement: recover one row of the `x` array and the associated Jacobian entry, corresponding to `channel`, from incoming and outgoing momenta. Also compute the phase-space volume.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_recover_channel
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_recover_channel &
        (forest, channel, sqrts, x, phs_factor, volume)
      type(phs_forest_t), intent(inout) :: forest
      integer, intent(in) :: channel
      real(default), intent(in) :: sqrts
      real(default), dimension(:,:), intent(inout) :: x
      real(default), dimension(:), intent(inout) :: phs_factor
      real(default), intent(out) :: volume
      integer :: g, t
      integer(TC) :: k, k_in
      g = forest%grove_lookup (channel)
      t = channel - forest%grove(g)%tree_count_offset
      call phs_prt_set_undefined (forest%prt)
      k_in = forest%n_tot
      forall (k = 1:forest%n_in)
         forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
      end forall
      forall (k = 1:forest%n_out)
         forest%prt(ibset(0,k-1)) = forest%prt_out(k)
      end forall
      call phs_forest_combine_particles (forest)
      call phs_tree_compute_volume &
          (forest%grove(g)%tree(t), sqrts, volume)
      call phs_tree_compute_x_from_momenta &
          (forest%grove(g)%tree(t), &
           forest%prt, phs_factor(channel), sqrts, x(:,channel))
    end subroutine phs_forest_recover_channel
```

11.6.13 Unit tests

Write a possible phase-space file for a $2 \rightarrow 3$ process and make the corresponding forest, print the forest. Choose some in-particle momenta and a random-number array and evaluate out-particles and phase-space factors.

⟨*PHS forests: public*⟩+≡
```
    public :: phs_forest_test
```

⟨*PHS forests: procedures*⟩+≡
```
    subroutine phs_forest_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```

⟨*PHS forests: execute tests*⟩
```
    end subroutine phs_forest_test
```


⟨*PHS forests: execute tests*⟩≡
```
    call test (phs_forest_1, "phs_forest_1", &
         "check phs forest setup", &
         u, results)
```
⟨*PHS forests: tests*⟩≡
```
    subroutine phs_forest_1 (u)
      use os_interface
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(phs_forest_t) :: forest
      type(model_list_t) :: model_list
      type(phs_channel_t), dimension(:), allocatable :: channel
      type(model_t), pointer :: model => null ()
      type(string_t) :: process_id
      type(flavor_t), dimension(5) :: flv
      type(string_t) :: filename
      type(interaction_t) :: int
      integer, parameter :: unit_fix = 20
      type(mapping_defaults_t) :: mapping_defaults
      logical :: found_process, ok
      integer :: n_channel, ch, i
      logical, dimension(4) :: active = .true.
      real(default) :: sqrts = 1000
      real(default), dimension(5,4) :: x
      real(default), dimension(4) :: factor
      real(default) :: volume

      write (u, "(A)")  "* Test output: PHS forest"
      write (u, "(A)")  "*   Purpose: test PHS forest routines"
      write (u, "(A)")

      write (u, "(A)")  "* Reading model file"

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
           (var_str("SM"), var_str("SM.mdl"), os_data, model)
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Create phase-space file 'phs_forest_test.phs'"
      write (u, "(A)")

      call flavor_init (flv, [11, -11, 11, -11, 22], model)
      open (file="phs_forest_test.phs", unit=unit_fix, action="write")
      write (unit_fix, *) "process foo"
      write (unit_fix, *) 'md5sum_process    = "6ABA33BC2927925D0F073B1C1170780A"'
      write (unit_fix, *) 'md5sum_model_par  = "1A0B151EE6E2DEB92D880320355A3EAB"'
      write (unit_fix, *) 'md5sum_phs_config = "B6A8877058809A8BDD54753CDAB83ACE"'
      write (unit_fix, *) "sqrts          =    100.00000000000000"
```

```
write (unit_fix, *) "m_threshold_s =    50.000000000000000"
write (unit_fix, *) "m_threshold_t =    100.00000000000000"
write (unit_fix, *) "off_shell =            2"
write (unit_fix, *) "t_channel =            6"
write (unit_fix, *) "keep_nonresonant =  F"
write (unit_fix, *) ""
write (unit_fix, *) "  grove"
write (unit_fix, *) "    tree 3 7"
write (unit_fix, *) "      map 3 s_channel 23"
write (unit_fix, *) "    tree 5 7"
write (unit_fix, *) "    tree 6 7"
write (unit_fix, *) "  grove"
write (unit_fix, *) "    tree 9 11"
write (unit_fix, *) "      map 9 t_channel 22"
close (unit_fix)

write (u, "(A)")
write (u, "(A)")  "* Read phase-space file 'phs_forest_test.phs'"

call syntax_phs_forest_init ()
process_id = "foo"
filename = "phs_forest_test.phs"
call phs_forest_read &
     (forest, filename, process_id, 2, 3, model, found_process)

write (u, "(A)")
write (u, "(A)")  "* Set parameters, flavors, equiv, momenta"
write (u, "(A)")

call phs_forest_set_flavors (forest, flv)
call phs_forest_set_parameters (forest, mapping_defaults, .false.)
call phs_forest_setup_prt_combinations (forest)
call phs_forest_set_equivalences (forest)
call interaction_init (int, 2, 0, 3)
call interaction_set_momentum (int, &
     vector4_moving (500._default, 500._default, 3), 1)
call interaction_set_momentum (int, &
     vector4_moving (500._default,-500._default, 3), 2)
call phs_forest_set_prt_in (forest, int)
n_channel = 2
x = 0
x(:,n_channel) = [0.3, 0.4, 0.1, 0.9, 0.6]
write (u, "(A)")  "   Input values:"
write (u, "(3x,5(1x," // FMT_12 // "))")  x(:,n_channel)

write (u, "(A)")
write (u, "(A)")  "* Evaluating phase space"

call phs_forest_evaluate_selected_channel (forest, &
     n_channel, active, sqrts, x, factor, volume, ok)
call phs_forest_evaluate_other_channels (forest, &
     n_channel, active, sqrts, x, factor, combine=.true.)
call phs_forest_get_prt_out (forest, int)
write (u, "(A)")  "   Output values:"
```

```
      do ch = 1, 4
         write (u, "(3x,5(1x," // FMT_12 // "))")  x(:,ch)
      end do
      call interaction_write (int, u)
      write (u, "(A)")  "  Factors:"
      write (u, "(3x,5(1x," // FMT_12 // "))")  factor
      write (u, "(A)")  "  Volume:"
      write (u, "(3x,5(1x," // FMT_12 // "))")  volume
      call phs_forest_write (forest, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compute equivalences"

      n_channel = 4
      allocate (channel (n_channel))
      call phs_forest_get_equivalences (forest, &
           channel, .true.)
      do i = 1, n_channel
         write (u, "(1x,I0,':')", advance = "no")  ch
         call channel(i)%write (u)
      end do

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call model_list%final ()
      call phs_forest_final (forest)
      call syntax_phs_forest_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_forest_1"

  end subroutine phs_forest_1
```

11.7 Finding phase space parameterizations

If the phase space configuration is not found in the appropriate file, we should
generate one.

The idea is to construct all Feynman diagrams subject to certain constraints
which eliminate everything that is probably irrelevant for the integration. These
Feynman diagrams (cascades) are grouped in groves by finding equivalence
classes related by symmetry and ordered with respect to their importance (resonances). Finally, the result (or part of it) is written to file and used for the
integration.

This module may eventually disappear and be replaced by CAML code. In
particular, we need here a set of Feynman rules (vertices with particle codes,
but not the factors). Thus, the module works for the Standard Model only.

Note that this module is stand-alone, it communicates to the main program
only via the generated ASCII phase-space configuration file.

⟨cascades.f90⟩≡

⟨*File header*⟩

```
module cascades
```

⟨*Use kinds*⟩
```
  use kinds, only: TC, i8, i32 !NODEP!
```
⟨*Use strings*⟩
```
  use limits, only: CASCADE_SET_FILL_RATIO, MAX_WARN_RESONANCE !NODEP!
  use limits, only: FMT_12, FMT_19 !NODEP!
```
⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use unit_tests
  use hashes
  use sorting
  use pdg_arrays, only: UNDEFINED
  use models
  use flavors
  use phs_forests
```

⟨*Standard module head*⟩

⟨*Cascades: public*⟩

⟨*Cascades: parameters*⟩

⟨*Cascades: types*⟩

⟨*Cascades: interfaces*⟩

```
contains
```

⟨*Cascades: procedures*⟩

⟨*Cascades: tests*⟩

```
end module cascades
```

11.7.1 The mapping modes

The valid mapping modes, to be used below. We will make use of the convention that mappings of internal particles have a positive value. Only for positive values, the flavor code is propagated when combining cascades.

⟨*Mapping modes*⟩≡
```
    integer, parameter :: &
        & EXTERNAL_PRT = -1, &
        & NO_MAPPING = 0, S_CHANNEL = 1, T_CHANNEL =  2, U_CHANNEL = 3, &
        & RADIATION = 4, COLLINEAR = 5, INFRARED = 6, &
        & STEP_MAPPING_E = 11, STEP_MAPPING_H = 12, &
        & ON_SHELL = 99
```

⟨*Cascades: parameters*⟩≡
  ⟨*Mapping modes*⟩

11.7.2 The cascade type

A cascade is essentially the same as a decay tree (both definitions may be merged in a later version). It contains a linked tree of nodes, each of which representing an internal particle. In contrast to decay trees, each node has a definite particle code. These nodes need not be modified, therefore we can use pointers and do not have to copy them. Thus, physically each cascades has only a single node, the mother particle. However, to be able to compare trees quickly, we store in addition an array of binary codes which is always sorted in ascending order. This is accompanied by a corresponding list of particle codes. The index is the location of the corresponding cascade in the cascade set, this may be used to access the daughters directly.

The real mass is the particle mass belonging to the particle code. The minimal mass is the sum of the real masses of all its daughters; this is the kinematical cutoff. The effective mass may be zero if the particle mass is below a certain threshold; it may be the real mass if the particle is resonant; or it may be some other value.

The logical `t_channel` is set if this a t-channel line, while `initial` is true only for an initial particle. Note that both initial particles are also `t_channel` by definition, and that they are distinguished by the direction of the tree: One of them decays and is the root of the tree, while the other one is one of the leaves.

The cascade is a list of nodes (particles) which are linked via the `daughter` entries. The node is the mother particle of the decay cascade. Much of the information in the nodes is repeated in arrays, to be accessible more easily. The arrays will be kept sorted by binary codes.

The counter `n_off_shell` is increased for each internal line that is neither resonant nor log-enhanced. It is set to zero if the current line is resonant, since this implies on-shell particle production and subsequent decay.

The counter `n_t_channel` is non-negative once an initial particle is included in the tree: then, it counts the number of t-channel lines.

The `multiplicity` is the number of branchings to follow until all daughters are on-shell. A resonant or non-decaying particle has multiplicity one. Merging nodes, the multiplicities add unless the mother is a resonance. An initial or final node has multiplicity zero.

The arrays correspond to the subnode tree `tree` of the current cascade. PDG codes are stored only for those positions which are resonant, with the exception of the last entry, i.e., the current node. Other positions, in particular external legs, are assigned undefined PDG code.

A cascade is uniquely identified by its tree, the tree of PDG codes, and the tree of mappings. The tree of resonances is kept only to mask the PDG tree as described above.

⟨*Cascades: types*⟩≡

```
type :: cascade_t
   private
   ! counters
   integer :: index = 0
   integer :: grove = 0
   ! status
   logical :: active = .false.
   logical :: complete = .false.
```

```
        logical :: incoming = .false.
        ! this node
        integer(TC) :: bincode = 0
        type(flavor_t) :: flv
        integer :: pdg = UNDEFINED
        logical :: is_vector = .false.
        real(default) :: m_min = 0
        real(default) :: m_rea = 0
        real(default) :: m_eff = 0
        integer :: mapping = NO_MAPPING
        logical :: on_shell = .false.
        logical :: resonant = .false.
        logical :: log_enhanced = .false.
        logical :: t_channel = .false.
        ! global tree properties
        integer :: multiplicity = 0
        integer :: internal = 0
        integer :: n_off_shell = 0
        integer :: n_resonances = 0
        integer :: n_log_enhanced = 0
        integer :: n_t_channel = 0
        integer :: res_hash = 0
        ! the sub-node tree
        integer :: depth = 0
        integer(TC), dimension(:), allocatable :: tree
        integer, dimension(:), allocatable :: tree_pdg
        integer, dimension(:), allocatable :: tree_mapping
        logical, dimension(:), allocatable :: tree_resonant
        ! branch connections
        logical :: has_children = .false.
        type(cascade_t), pointer :: daughter1 => null ()
        type(cascade_t), pointer :: daughter2 => null ()
        type(cascade_t), pointer :: mother => null ()
        ! next in list
        type(cascade_t), pointer :: next => null ()
      end type cascade_t
```

⟨*Cascades: procedures*⟩≡
```
    subroutine cascade_init (cascade, depth)
      type(cascade_t), intent(out) :: cascade
      integer, intent(in) :: depth
      integer, save :: index = 0
      index = cascade_index ()
      cascade%index = index
      cascade%depth = depth
      cascade%active = .true.
      allocate (cascade%tree (depth))
      allocate (cascade%tree_pdg (depth))
      allocate (cascade%tree_mapping (depth))
      allocate (cascade%tree_resonant (depth))
    end subroutine cascade_init
```

Keep and increment a global index

⟨*Cascades: procedures*⟩+≡

```
      function cascade_index (seed) result (index)
        integer :: index
        integer, intent(in), optional :: seed
        integer, save :: i = 0
        if (present (seed))  i = seed
        i = i + 1
        index = i
      end function cascade_index
```

We need three versions of writing cascades. This goes to the phase-space file:

⟨*Cascades: procedures*⟩+≡

```
      subroutine cascade_write_file_format (cascade, model, unit)
        type(cascade_t), intent(in) :: cascade
        type(model_t), intent(in), target :: model
        integer, intent(in), optional :: unit
        type(flavor_t) :: flv
        integer :: u, i
1       format(3x,A,1x,40(1x,I4))
2       format(3x,A,1x,I3,1x,A,1x,I7,1x,'!',1x,A)
        u = output_unit (unit);  if (u < 0)  return
        call write_reduced (cascade%tree, u)
        write (u, "(A)")
        do i = 1, cascade%depth
           call flavor_init (flv, cascade%tree_pdg(i), model)
           select case (cascade%tree_mapping(i))
           case (NO_MAPPING, EXTERNAL_PRT)
           case (S_CHANNEL)
              write(u,2) 'map', &
                   cascade%tree(i), 's_channel', abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
           case (T_CHANNEL)
              write(u,2) 'map', &
                   cascade%tree(i), 't_channel', abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
           case (U_CHANNEL)
              write(u,2) 'map', &
                   cascade%tree(i), 'u_channel', abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
           case (RADIATION)
              write(u,2) 'map', &
                   cascade%tree(i), 'radiation', abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
           case (COLLINEAR)
              write(u,2) 'map', &
                   cascade%tree(i), 'collinear', abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
           case (INFRARED)
              write(u,2) 'map', &
                   cascade%tree(i), 'infrared ',  abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
           case (ON_SHELL)
              write(u,2) 'map', &
                   cascade%tree(i), 'on_shell ', abs (cascade%tree_pdg(i)), &
                   char (flavor_get_name (flv))
```

```
            case default
               call msg_bug (" Impossible mapping mode encountered")
            end select
         end do
   contains
      subroutine write_reduced (array, unit)
         integer(TC), dimension(:), intent(in) :: array
         integer, intent(in) :: unit
         integer :: i
         write (u, "(3x,A,1x)", advance="no")  "tree"
         do i = 1, size (array)
            if (decay_level (array(i)) > 1) then
               write (u, "(1x,I0)", advance="no")  array(i)
            end if
         end do
      end subroutine write_reduced

      elemental function decay_level (k) result (l)
         integer(TC), intent(in) :: k
         integer :: l
         integer :: i
         l = 0
         do i = 0, bit_size(k) - 1
            if (btest(k,i)) l = l + 1
         end do
      end function decay_level
      subroutine start_comment (u)
         integer, intent(in) :: u
         write(u, '(1x,A)', advance='no') '!'
      end subroutine start_comment
   end subroutine cascade_write_file_format
```

This creates metapost source for graphical display:

⟨*Cascades: procedures*⟩+≡

```
      subroutine cascade_write_graph_format (cascade, count, unit)
         type(cascade_t), intent(in) :: cascade
         integer, intent(in) :: count
         integer, intent(in), optional :: unit
         integer :: u
         integer(TC) :: mask
         type(string_t) :: left_str, right_str
         u = output_unit (unit);  if (u < 0)  return
         mask = 2**((cascade%depth+3)/2) - 1
         left_str = ""
         right_str = ""
         write (u, '(A)') "\begin{minipage}{105pt}"
         write (u, '(A)') "\vspace{30pt}"
         write (u, '(A)') "\begin{center}"
         write (u, '(A)') "\begin{fmfgraph*}(55,55)"
         call graph_write (cascade, mask)
         write (u, '(A)') "\fmfleft{" // char (extract (left_str, 2)) // "}"
         write (u, '(A)') "\fmfright{" // char (extract (right_str, 2)) // "}"
         write (u, '(A)') "\end{fmfgraph*}\\"
         write (u, '(A,I5,A)') "\fbox{$", count, "$}"
```

```fortran
      write (u, '(A)') "\end{center}"
      write (u, '(A)') "\end{minipage}"
      write (u, '(A)') "%"
    contains
      recursive subroutine graph_write (cascade, mask, reverse)
        type(cascade_t), intent(in) :: cascade
        integer(TC), intent(in) :: mask
        logical, intent(in), optional :: reverse
        logical :: rev
        rev = .false.;  if (present(reverse))  rev = reverse
        if (cascade%has_children) then
           if (.not.rev) then
              call vertex_write (cascade, cascade%daughter1, mask)
              call vertex_write (cascade, cascade%daughter2, mask)
           else
              call vertex_write (cascade, cascade%daughter2, mask, .true.)
              call vertex_write (cascade, cascade%daughter1, mask, .true.)
           end if
           if (cascade%complete) then
              call vertex_write (cascade, cascade%mother, mask, .true.)
              write (u, '(A,I0,A)') "\fmfv{d.shape=square}{v0}"
           end if
        else
           if (cascade%incoming) then
              call external_write (cascade%bincode, &
                   flavor_get_tex_name (flavor_anti (cascade%flv)), left_str)
           else
              call external_write (cascade%bincode, &
                   flavor_get_tex_name (cascade%flv), right_str)
           end if
        end if
      end subroutine graph_write
      recursive subroutine vertex_write (cascade, daughter, mask, reverse)
        type(cascade_t), intent(in) :: cascade, daughter
        integer(TC), intent(in) :: mask
        logical, intent(in), optional :: reverse
        integer :: bincode
        if (cascade%complete) then
           bincode = 0
        else
           bincode = cascade%bincode
        end if
        call graph_write (daughter, mask, reverse)
        if (daughter%has_children) then
           call line_write (bincode, daughter%bincode, daughter%flv, &
                mapping=daughter%mapping)
        else
           call line_write (bincode, daughter%bincode, daughter%flv)
        end if
      end subroutine vertex_write
      subroutine line_write (i1, i2, flv, mapping)
        integer(TC), intent(in) :: i1, i2
        type(flavor_t), intent(in) :: flv
        integer, intent(in), optional :: mapping
```

```fortran
    integer :: k1, k2
    type(string_t) :: prt_type
    select case (flavor_get_spin_type (flv))
    case (SCALAR);        prt_type = "plain"
    case (SPINOR);        prt_type = "fermion"
    case (VECTOR);        prt_type = "boson"
    case (VECTORSPINOR); prt_type = "fermion"
    case (TENSOR);        prt_type = "dbl_wiggly"
    case default;         prt_type = "dashes"
    end select
    if (flavor_is_antiparticle (flv)) then
       k1 = i2;  k2 = i1
    else
       k1 = i1;  k2 = i2
    end if
    if (present (mapping)) then
       select case (mapping)
       case (S_CHANNEL)
          write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & ",f=blue,lab=\sm\blue$" // &
               & char (flavor_get_tex_name (flv)) // "$}" // &
               & "{v", k1, ",v", k2, "}"
       case (T_CHANNEL, U_CHANNEL)
          write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & ",f=cyan,lab=\sm\cyan$" // &
               & char (flavor_get_tex_name (flv)) // "$}" // &
               & "{v", k1, ",v", k2, "}"
       case (RADIATION)
          write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & ",f=green,lab=\sm\green$" // &
               & char (flavor_get_tex_name (flv)) // "$}" // &
               & "{v", k1, ",v", k2, "}"
       case (COLLINEAR)
          write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & ",f=magenta,lab=\sm\magenta$" // &
               & char (flavor_get_tex_name (flv)) // "$}" // &
               & "{v", k1, ",v", k2, "}"
       case (INFRARED)
          write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & ",f=red,lab=\sm\red$" // &
               & char (flavor_get_tex_name (flv)) // "$}" // &
               & "{v", k1, ",v", k2, "}"
       case default
          write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & ",f=black}" // &
               & "{v", k1, ",v", k2, "}"
       end select
    else
       write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
               & "}" // &
               & "{v", k1, ",v", k2, "}"
    end if
  end subroutine line_write
  subroutine external_write (bincode, name, ext_str)
```

```
      integer(TC), intent(in) :: bincode
      type(string_t), intent(in) :: name
      type(string_t), intent(inout) :: ext_str
      character(len=20) :: str
      write (str, '(A2,I0)') ",v", bincode
      ext_str = ext_str // trim (str)
      write (u, '(A,I0,A,I0,A)') "\fmflabel{\sm$" &
        // char (name) &
        // "\,(", bincode, ")" &
        // "$}{v", bincode, "}"
    end subroutine external_write
  end subroutine cascade_write_graph_format
```

This is for screen/debugging output:

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_write (cascade, unit)
    type(cascade_t), intent(in) :: cascade
    integer, intent(in), optional :: unit
    integer :: u
    character(9) :: depth
    u = output_unit (unit);  if (u < 0)  return
    write (u, "(A,(1x,I7))") 'Cascade #', cascade%index
    write (u, "(A,(1x,I7))") ' Grove:        #', cascade%grove
    write (u, "(A,3(1x,L1))") ' act/cmp/inc: ', &
        cascade%active, cascade%complete, cascade%incoming
    write (u, "(A,I0)") ' Bincode:       ', cascade%bincode
    write (u, "(A)", advance="no") ' Flavor:       '
    call flavor_write (cascade%flv, unit)
    write (u, "(A,I9)") ' Active flavor:', cascade%pdg
    write (u, "(A,L1)") ' Is vector:    ', cascade%is_vector
    write (u, "(A,3(1x," // FMT_19 // "))") ' Mass (m/r/e): ', &
        cascade%m_min, cascade%m_rea, cascade%m_eff
    write (u, "(A,I1)") ' Mapping:      ', cascade%mapping
    write (u, "(A,3(1x,L1))") ' res/log/tch: ', &
        cascade%resonant, cascade%log_enhanced, cascade%t_channel
    write (u, "(A,(1x,I7))") ' Multiplicity: ', cascade%multiplicity
    write (u, "(A,2(1x,I7))") ' n intern/off: ', &
        cascade%internal, cascade%n_off_shell
    write (u, "(A,3(1x,I7))") ' n res/log/tch:', &
        cascade%n_resonances, cascade%n_log_enhanced, cascade%n_t_channel
    write (u, "(A,I7)") ' Depth:        ', cascade%depth
    write (depth, "(I7)") cascade%depth
    write (u, "(A," // depth // "(1x,I7))") &
      ' Tree:         ', cascade%tree
    write (u, "(A," // depth // "(1x,I7))") &
      ' Tree(PDG):    ', cascade%tree_pdg
    write (u, "(A," // depth // "(1x,I7))") &
      ' Tree(mapping):', cascade%tree_mapping
    write (u, "(A," // depth // "(1x,L1))") &
      ' Tree(res):    ', cascade%tree_resonant
    if (cascade%has_children) then
       write (u, "(A,I7,1x,I7)") ' Daughter1/2:  ', &
           cascade%daughter1%index, cascade%daughter2%index
    end if
```

```
          if (associated (cascade%mother)) then
             write (u, "(A,I7)") '  Mother:        ', cascade%mother%index
          end if
       end subroutine cascade_write
```

11.7.3 Creating new cascades

This initializes a single-particle cascade (external, final state). The PDG entry
in the tree is set undefined because the cascade is not resonant. However, the
flavor entry is set, so the cascade flavor is identified nevertheless.

⟨*Cascades: procedures*⟩+≡
```
       subroutine cascade_init_outgoing (cascade, flv, pos, m_thr)
         type(cascade_t), intent(out) :: cascade
         type(flavor_t), intent(in) :: flv
         integer, intent(in) :: pos
         real(default), intent(in) :: m_thr
         call cascade_init (cascade, 1)
         cascade%bincode = ibset (0_TC, pos-1)
         cascade%flv = flv
         cascade%pdg = abs (flavor_get_pdg (cascade%flv))
         cascade%is_vector = flavor_get_spin_type (flv) == VECTOR
         cascade%m_min = flavor_get_mass (flv)
         cascade%m_rea = cascade%m_min
         if (cascade%m_rea >= m_thr) then
            cascade%m_eff = cascade%m_rea
         end if
         cascade%on_shell = .true.
         cascade%multiplicity = 1
         cascade%tree(1) = cascade%bincode
         cascade%tree_pdg(1) = cascade%pdg
         cascade%tree_mapping(1) = EXTERNAL_PRT
         cascade%tree_resonant(1) = .false.
       end subroutine cascade_init_outgoing
```

The same for an incoming line:

⟨*Cascades: procedures*⟩+≡
```
       subroutine cascade_init_incoming (cascade, flv, pos, m_thr)
         type(cascade_t), intent(out) :: cascade
         type(flavor_t), intent(in) :: flv
         integer, intent(in) :: pos
         real(default), intent(in) :: m_thr
         call cascade_init (cascade, 1)
         cascade%incoming = .true.
         cascade%bincode = ibset (0_TC, pos-1)
         cascade%flv = flavor_anti (flv)
         cascade%pdg = abs (flavor_get_pdg (flv))
         cascade%is_vector = flavor_get_spin_type (flv) == VECTOR
         cascade%m_min = flavor_get_mass (flv)
         cascade%m_rea = cascade%m_min
         if (cascade%m_rea >= m_thr) then
            cascade%m_eff = cascade%m_rea
         end if
```

```
      cascade%on_shell = .true.
      cascade%n_t_channel = 0
      cascade%n_off_shell = 0
      cascade%tree(1) = cascade%bincode
      cascade%tree_pdg(1) = cascade%pdg
      cascade%tree_mapping(1) = EXTERNAL_PRT
      cascade%tree_resonant(1) = .false.
    end subroutine cascade_init_incoming
```

11.7.4 Tools

This function returns true if the two cascades share no common external particle. This is a requirement for joining them.

⟨*Cascades: interfaces*⟩≡
```
    interface operator(.disjunct.)
       module procedure cascade_disjunct
    end interface
```

⟨*Cascades: procedures*⟩+≡
```
    function cascade_disjunct (cascade1, cascade2) result (flag)
      logical :: flag
      type(cascade_t), intent(in) :: cascade1, cascade2
      flag = iand (cascade1%bincode, cascade2%bincode) == 0
    end function cascade_disjunct
```

Compute a hash code for the resonance pattern of a cascade. We count the number of times each particle appears as a resonance.

We pack the PDG codes of the resonances in two arrays (s-channel and t-channel), sort them both, concatenate the results, transfer to `i8` integers, and compute the hash code from this byte stream.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_assign_resonance_hash (cascade)
      type(cascade_t), intent(inout) :: cascade
      integer(i8), dimension(1) :: mold
      cascade%res_hash = hash (transfer &
          (concat (sort (pack (cascade%tree_pdg, &
                               cascade%tree_resonant)), &
                   sort (pack (cascade%tree_pdg, &
                               cascade%tree_mapping == T_CHANNEL .or. &
                               cascade%tree_mapping == U_CHANNEL))), &
          mold))
    end subroutine cascade_assign_resonance_hash
```

11.7.5 Hash entries for cascades

We will set up a hash array which contains keys of and pointers to cascades. We hold a list of cascade (pointers) within each bucket. This is not for collision resolution, but for keeping similar, but unequal cascades together.

⟨*Cascades: types*⟩+≡

```
    type :: cascade_p
       type(cascade_t), pointer :: cascade => null ()
       type(cascade_p), pointer :: next => null ()
    end type cascade_p
```

Here is the bucket or hash entry type:

⟨*Cascades: types*⟩+≡

```
    type :: hash_entry_t
       integer(i32) :: hashval = 0
       integer(i8), dimension(:), allocatable :: key
       type(cascade_p), pointer :: first => null ()
       type(cascade_p), pointer :: last => null ()
    end type hash_entry_t
```

Finalize: just deallocate the list; the contents are just pointers.

⟨*Cascades: procedures*⟩+≡

```
    subroutine hash_entry_final (hash_entry)
      type(hash_entry_t), intent(inout) :: hash_entry
      type(cascade_p), pointer :: current
      do while (associated (hash_entry%first))
         current => hash_entry%first
         hash_entry%first => current%next
         deallocate (current)
      end do
    end subroutine hash_entry_final
```

Output: concise format for debugging, just list cascade indices.

⟨*Cascades: procedures*⟩+≡

```
    subroutine hash_entry_write (hash_entry, unit)
      type(hash_entry_t), intent(in) :: hash_entry
      integer, intent(in), optional :: unit
      type(cascade_p), pointer :: current
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(1x,A)", advance="no")  "Entry:"
      do i = 1, size (hash_entry%key)
         write (u, "(1x,I0)", advance="no")  hash_entry%key(i)
      end do
      write (u, "(1x,A)", advance="no")  "->"
      current => hash_entry%first
      do while (associated (current))
         write (u, "(1x,I7)", advance="no") current%cascade%index
         current => current%next
      end do
      write (u, *)
    end subroutine hash_entry_write
```

This function adds a cascade pointer to the bucket. If `ok` is present, check first if it is already there and return failure if yes. If `cascade_ptr` is also present, set it to the current cascade if successful. If not, set it to the cascade that is already there.

⟨*Cascades: procedures*⟩+≡

```
subroutine hash_entry_add_cascade_ptr (hash_entry, cascade, ok, cascade_ptr)
  type(hash_entry_t), intent(inout) :: hash_entry
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out), optional :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  type(cascade_p), pointer :: current
  if (present (ok)) then
     call hash_entry_check_cascade (hash_entry, cascade, ok, cascade_ptr)
     if (.not. ok)  return
  end if
  allocate (current)
  current%cascade => cascade
  if (associated (hash_entry%last)) then
     hash_entry%last%next => current
  else
     hash_entry%first => current
  end if
  hash_entry%last => current
end subroutine hash_entry_add_cascade_ptr
```

This function checks whether a cascade is already in the bucket. For incomplete cascades, we look for an exact match. It should suffice to verify the tree, the PDG codes, and the mapping modes. This is the information that is written to the phase space file.

For complete cascades, we ignore the PDG code at positions with mappings infrared, collinear, or t/u-channel. Thus a cascade which is distinguised only by PDG code at such places, is flagged existent. If the convention is followed that light particles come before heavier ones (in the model definition), this ensures that the lightest particle is kept in the appropriate place, corresponding to the strongest peak.

For external cascades (incoming/outgoing) we take the PDG code into account even though it is zeroed in the PDG-code tree.

⟨Cascades: procedures⟩+≡
```
subroutine hash_entry_check_cascade (hash_entry, cascade, ok, cascade_ptr)
  type(hash_entry_t), intent(in), target :: hash_entry
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out) :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  type(cascade_p), pointer :: current
  integer, dimension(:), allocatable :: tree_pdg
  ok = .true.
  allocate (tree_pdg (size (cascade%tree_pdg)))
  if (cascade%complete) then
     where (cascade%tree_mapping == INFRARED .or. &
            cascade%tree_mapping == COLLINEAR .or. &
            cascade%tree_mapping == T_CHANNEL .or. &
            cascade%tree_mapping == U_CHANNEL)
       tree_pdg = 0
     elsewhere
       tree_pdg = cascade%tree_pdg
     end where
  else
     tree_pdg = cascade%tree_pdg
```

```
         end if
         current => hash_entry%first
         do while (associated (current))
            if (current%cascade%depth == cascade%depth) then
               if (all (current%cascade%tree == cascade%tree)) then
                  if (all (current%cascade%tree_mapping == cascade%tree_mapping)) &
                       then
                     if (all (current%cascade%tree_pdg .match. tree_pdg)) then
                        if (present (cascade_ptr))  cascade_ptr => current%cascade
                        ok = .false.;  return
                     end if
                  end if
               end if
            end if
            current => current%next
         end do
         if (present (cascade_ptr))  cascade_ptr => cascade
      end subroutine hash_entry_check_cascade
```

For PDG codes, we specify that the undefined code matches any code. This is
already defined for flavor objects, but here we need it for the codes themselves.

⟨*Cascades: interfaces*⟩+≡
```
      interface operator(.match.)
         module procedure pdg_match
      end interface
```

⟨*Cascades: procedures*⟩+≡
```
      elemental function pdg_match (pdg1, pdg2) result (flag)
        logical :: flag
        integer(TC), intent(in) :: pdg1, pdg2
        select case (pdg1)
        case (0)
           flag = .true.
        case default
           select case (pdg2)
           case (0)
              flag = .true.
           case default
              flag = pdg1 == pdg2
           end select
        end select
      end function pdg_match
```

11.7.6   The cascade set

The cascade set will later be transformed into the decay forest. It is set up as a
linked list. In addition to the usual `first` and `last` pointers, there is a `first_t`
pointer which points to the first t-channel cascade (after all s-channel cascades),
and a `first_k` pointer which points to the first final cascade (with a keystone).

As an auxiliary device, the object contains a hash array with associated
parameters where an additional pointer is stored for each cascade. The keys are

made from the relevant cascade data. This hash is used for fast detection (and thus avoidance) of double entries in the cascade list.

⟨*Cascades: public*⟩≡
```
    public :: cascade_set_t
```

⟨*Cascades: types*⟩+≡
```
    type :: cascade_set_t
      private
      type(model_t), pointer :: model
      integer :: n_in, n_out, n_tot
      type(flavor_t), dimension(:,:), allocatable :: flv
      integer :: depth_out, depth_tot
      real(default) :: sqrts = 0
      real(default) :: m_threshold_s = 0
      real(default) :: m_threshold_t = 0
      integer :: off_shell = 0
      integer :: t_channel = 0
      logical :: keep_nonresonant
      integer :: n_groves = 0
      ! The cascade list
      type(cascade_t), pointer :: first => null ()
      type(cascade_t), pointer :: last => null ()
      type(cascade_t), pointer :: first_t => null ()
      type(cascade_t), pointer :: first_k => null ()
      ! The hashtable
      integer :: n_entries = 0
      real :: fill_ratio = 0
      integer :: n_entries_max = 0
      integer(i32) :: mask = 0
      logical :: fatal_beam_decay = .true.
      type(hash_entry_t), dimension(:), allocatable :: entry
    end type cascade_set_t
```

Return true if there are cascades which are active and complete, so the phase space file would be nonempty.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_is_valid
```

⟨*Cascades: procedures*⟩+≡
```
    function cascade_set_is_valid (cascade_set) result (flag)
      logical :: flag
      type(cascade_set_t), intent(in) :: cascade_set
      type(cascade_t), pointer :: cascade
      flag = .false.
      cascade => cascade_set%first_k
      do while (associated (cascade))
         if (cascade%active .and. cascade%complete) then
            flag = .true.
            return
         end if
         cascade => cascade%next
      end do
    end function cascade_set_is_valid
```

The initializer sets up the hash table with some initial size guessed by looking at the number of external particles. We choose 256 for 3 external particles and a factor of 4 for each additional particle, limited at 2^{30}=1G.

⟨*Limits: public parameters*⟩+≡
```
    real, parameter, public :: CASCADE_SET_FILL_RATIO = 0.1
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
         fatal_beam_decay, flv)
      type(cascade_set_t), intent(out) :: cascade_set
      type(model_t), intent(in), target :: model
      integer, intent(in) :: n_in, n_out
      type(phs_parameters_t), intent(in) :: phs_par
      logical, intent(in) :: fatal_beam_decay
      type(flavor_t), dimension(:,:), intent(in), optional :: flv
      integer :: size_guess
      cascade_set%model => model
      cascade_set%n_in = n_in
      cascade_set%n_out = n_out
      cascade_set%n_tot = n_in + n_out
      if (present (flv)) then
         allocate (cascade_set%flv (size (flv, 1), size (flv, 2)))
         call flavor_init (cascade_set%flv, flavor_get_pdg (flv), model)
      end if
      select case (n_in)
      case (1);  cascade_set%depth_out = 2 * n_out - 3
      case (2);  cascade_set%depth_out = 2 * n_out - 1
      end select
      cascade_set%depth_tot = 2 * cascade_set%n_tot - 3
      cascade_set%sqrts = phs_par%sqrts
      cascade_set%m_threshold_s = phs_par%m_threshold_s
      cascade_set%m_threshold_t = phs_par%m_threshold_t
      cascade_set%off_shell = phs_par%off_shell
      cascade_set%t_channel = phs_par%t_channel
      cascade_set%keep_nonresonant = phs_par%keep_nonresonant
      cascade_set%fill_ratio = CASCADE_SET_FILL_RATIO
      size_guess = ishft (256, min (2 * (cascade_set%n_tot - 3), 22))
      cascade_set%n_entries_max = size_guess * cascade_set%fill_ratio
      cascade_set%mask = size_guess - 1
      allocate (cascade_set%entry (0:cascade_set%mask))
      cascade_set%fatal_beam_decay = fatal_beam_decay
    end subroutine cascade_set_init
```

The finalizer has to delete both the hash and the list.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_final
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_final (cascade_set)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), pointer :: current
      integer :: i
      if (allocated (cascade_set%entry)) then
         do i = 0, cascade_set%mask
```

```
         call hash_entry_final (cascade_set%entry(i))
      end do
      deallocate (cascade_set%entry)
   end if
   do while (associated (cascade_set%first))
      current => cascade_set%first
      cascade_set%first => cascade_set%first%next
      deallocate (current)
   end do
end subroutine cascade_set_final
```

Write the process in ASCII format, in columns that are headed by the corresponding bincode.

⟨*Cascades: public*⟩+≡
```
public :: cascade_set_write_process_bincode_format
```

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_write_process_bincode_format (cascade_set, unit)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer, intent(in), optional :: unit
  integer, dimension(:), allocatable :: bincode, field_width
  integer :: n_in, n_out, n_tot, n_flv
  integer :: u, f, i, bc
  character(20) :: str
  type(string_t) :: fmt_head
  type(string_t), dimension(:), allocatable :: fmt_proc
  u = output_unit (unit);  if (u < 0)  return
  if (.not. allocated (cascade_set%flv)) return
  write (u, "('!',1x,A)")  "List of subprocesses with particle bincodes:"
  n_in  = cascade_set%n_in
  n_out = cascade_set%n_out
  n_tot = cascade_set%n_tot
  n_flv = size (cascade_set%flv, 2)
  allocate (bincode (n_tot), field_width (n_tot), fmt_proc (n_tot))
  bc = 1
  do i = 1, n_out
     bincode(n_in + i) = bc
     bc = 2 * bc
  end do
  do i = n_in, 1, -1
     bincode(i) = bc
     bc = 2 * bc
  end do
  do i = 1, n_tot
     write (str, "(I0)")  bincode(i)
     field_width(i) = len_trim (str)
     do f = 1, n_flv
        field_width(i) = max (field_width(i), &
             len (flavor_get_name (cascade_set%flv(i,f))))
     end do
  end do
  fmt_head = "('!'"
  do i = 1, n_tot
     fmt_head = fmt_head // ",1x,"
```

```
      fmt_proc(i) = "(1x,"
      write (str, "(I0)")  field_width(i)
      fmt_head = fmt_head // "I" // trim(str)
      fmt_proc(i) = fmt_proc(i) // "A" // trim(str)
      if (i == n_in) then
         fmt_head = fmt_head // ",1x,'  '"
      end if
   end do
   fmt_proc = fmt_proc // ")"
   fmt_head = fmt_head // ")"
   write (u, char (fmt_head))  bincode
   do f = 1, n_flv
      write (u, "('!')", advance="no")
      do i = 1, n_tot
         write (u, char (fmt_proc(i)), advance="no") &
               char (flavor_get_name (cascade_set%flv(i,f)))
         if (i == n_in)  write (u, "(1x,'=>')", advance="no")
      end do
      write (u, *)
   end do
   write (u, char (fmt_head))  bincode
end subroutine cascade_set_write_process_bincode_format
```

Write the process as a LaTeX expression.

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_write_process_tex_format (cascade_set, unit)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer, intent(in), optional :: unit
  integer :: u, f, i
  u = output_unit (unit);  if (u < 0)  return
  if (.not. allocated (cascade_set%flv)) return
  write (u, "(A)")  "\begin{align*}"
  do f = 1, size (cascade_set%flv, 2)
     do i = 1, cascade_set%n_in
        if (i > 1)  write (u, "(A)", advance="no") "\quad "
        write (u, "(A)", advance="no") &
              char (flavor_get_tex_name (cascade_set%flv(i,f)))
     end do
     write (u, "(A)", advance="no")  "\quad &\to\quad "
     do i = cascade_set%n_in + 1, cascade_set%n_tot
        if (i > cascade_set%n_in + 1)  write (u, "(A)", advance="no") "\quad "
        write (u, "(A)", advance="no") &
              char (flavor_get_tex_name (cascade_set%flv(i,f)))
     end do
     if (f < size (cascade_set%flv, 2)) then
        write (u, "(A)")  "\\"
     else
        write (u, "(A)")  ""
     end if
  end do
  write (u, "(A)")  "\end{align*}"
end subroutine cascade_set_write_process_tex_format
```
```

Three output routines: phase-space file, graph source code, and screen output.

This version generates the phase space file. It deals only with complete cascades.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_write_file_format
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_write_file_format (cascade_set, unit)
      type(cascade_set_t), intent(in), target :: cascade_set
      integer, intent(in), optional :: unit
      type(cascade_t), pointer :: cascade
      integer :: u, grove, count
      logical :: first_in_grove
      u = output_unit (unit); if (u < 0) return
      count = 0
      do grove = 1, cascade_set%n_groves
         first_in_grove = .true.
         cascade => cascade_set%first_k
         do while (associated (cascade))
            if (cascade%active .and. cascade%complete) then
               if (cascade%grove == grove) then
                  if (first_in_grove) then
                     first_in_grove = .false.
                     write (u, *)
                     write (u, "(1x,'!',1x,A,1x,I0,A)", advance='no') &
                        'Multiplicity =', cascade%multiplicity, ","
                     select case (cascade%n_resonances)
                     case (0)
                        write (u, '(1x,A)', advance='no') 'no resonances, '
                     case (1)
                        write (u, '(1x,A)', advance='no') '1 resonance, '
                     case default
                        write (u, '(1x,I0,1x,A)', advance='no') &
                             cascade%n_resonances, 'resonances, '
                     end select
                     write (u, '(1x,I0,1x,A)', advance='no') &
                          cascade%n_log_enhanced, 'logs, '
                     write (u, '(1x,I0,1x,A)', advance='no') &
                          cascade%n_off_shell, 'off-shell, '
                     select case (cascade%n_t_channel)
                     case (0); write (u, '(1x,A)') 's-channel graph'
                     case (1); write (u, '(1x,A)') '1 t-channel line'
                     case default
                        write(u,'(1x,I0,1x,A)') &
                             cascade%n_t_channel, 't-channel lines'
                     end select
                     write (u, '(1x,A,I0)') 'grove #', grove
                  end if
                  count = count + 1
                  write (u, "(1x,'!',1x,A,I0)") "Channel #", count
                  call cascade_write_file_format (cascade, cascade_set%model, u)
               end if
            end if
            cascade => cascade%next
```

1503

```
            end do
          end do
        end subroutine cascade_set_write_file_format
```

This is the graph output format, the driver-file

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_write_graph_format
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_write_graph_format &
        (cascade_set, filename, process_id, unit)
      type(cascade_set_t), intent(in), target :: cascade_set
      type(string_t), intent(in) :: filename, process_id
      integer, intent(in), optional :: unit
      type(cascade_t), pointer :: cascade
      integer :: u, grove, count, pgcount
      logical :: first_in_grove
      u = output_unit (unit);  if (u < 0)  return
      write (u, '(A)') "\documentclass[10pt]{article}"
      write (u, '(A)') "\usepackage{amsmath}"
      write (u, '(A)') "\usepackage{feynmp}"
      write (u, '(A)') "\usepackage{url}"
      write (u, '(A)') "\usepackage{color}"
      write (u, *)
      write (u, '(A)') "\textwidth 18.5cm"
      write (u, '(A)') "\evensidemargin -1.5cm"
      write (u, '(A)') "\oddsidemargin -1.5cm"
      write (u, *)
      write (u, '(A)') "\newcommand{\blue}{\color{blue}}"
      write (u, '(A)') "\newcommand{\green}{\color{green}}"
      write (u, '(A)') "\newcommand{\red}{\color{red}}"
      write (u, '(A)') "\newcommand{\magenta}{\color{magenta}}"
      write (u, '(A)') "\newcommand{\cyan}{\color{cyan}}"
      write (u, '(A)') "\newcommand{\sm}{\footnotesize}"
      write (u, '(A)') "\setlength{\parindent}{0pt}"
      write (u, '(A)') "\setlength{\parsep}{20pt}"
      write (u, *)
      write (u, '(A)') "\begin{document}"
      write (u, '(A)') "\begin{fmffile}{" // char (filename) // "}"
      write (u, '(A)') "\fmfcmd{color magenta; magenta = red + blue;}"
      write (u, '(A)') "\fmfcmd{color cyan; cyan = green + blue;}"
      write (u, '(A)') "\begin{fmfshrink}{0.5}"
      write (u, '(A)') "\begin{flushleft}"
      write (u, *)
      write (u, '(A)') "\noindent" // &
          & "\textbf{\large\texttt{WHIZARD} phase space channels}" // &
          & "\hfill\today"
      write (u, *)
      write (u, '(A)') "\vspace{10pt}"
      write (u, '(A)') "\noindent" // &
          & "\textbf{Process:} \url{" // char (process_id) // "}"
      call cascade_set_write_process_tex_format (cascade_set, u)
      write (u, *)
      write (u, '(A)') "\noindent" // &
```

1504

```
      & "\textbf{Note:} These are pseudo Feynman graphs that "
write (u, '(A)') "visualize phase-space parameterizations " // &
      & "(``integration channels'').  "
write (u, '(A)') "They do \emph{not} indicate Feynman graphs used for the " // &
      & "matrix element."
write (u, *)
write (u, '(A)') "\textbf{Color code:} " // &
      & "{\blue resonance,} " // &
      & "{\cyan t-channel,} " // &
      & "{\green radiation,} "
write (u, '(A)') "{\red infrared,} " // &
      & "{\magenta collinear,} " // &
      & "external/off-shell"
write (u, *)
write (u, '(A)') "\noindent" // &
      & "\textbf{Black square:} Keystone, indicates ordering of " // &
      & "phase space parameters."
write (u, *)
write (u, '(A)') "\vspace{-20pt}"
count = 0
pgcount = 0
do grove = 1, cascade_set%n_groves
   first_in_grove = .true.
   cascade => cascade_set%first
   do while (associated (cascade))
      if (cascade%active .and. cascade%complete) then
         if (cascade%grove == grove) then
            if (first_in_grove) then
               first_in_grove = .false.
               write (u, *)
               write (u, '(A)') "\vspace{20pt}"
               write (u, '(A)') "\begin{tabular}{l}"
               write (u, '(A,I5,A)') &
                    & "\fbox{\bf Grove \boldmath$", grove, "$} \\[10pt]"
               write (u, '(A,I1,A)') "Multiplicity: ", &
                    cascade%multiplicity, "\\"
               write (u, '(A,I1,A)') "Resonances:   ", &
                    cascade%n_resonances, "\\"
               write (u, '(A,I1,A)') "Log-enhanced: ", &
                    cascade%n_log_enhanced, "\\"
               write (u, '(A,I1,A)') "Off-shell:    ", &
                    cascade%n_off_shell, "\\"
               write (u, '(A,I1,A)') "t-channel:    ", &
                    cascade%n_t_channel, ""
               write (u, '(A)') "\end{tabular}"
            end if
            count = count + 1
            call cascade_write_graph_format (cascade, count, unit)
            if (pgcount >= 250) then
               write (u, '(A)') "\clearpage"
               pgcount = 0
            end if
         end if
      end if
```

```
              cascade => cascade%next
          end do
       end do
       write (u, '(A)') "\end{flushleft}"
       write (u, '(A)') "\end{fmfshrink}"
       write (u, '(A)') "\end{fmffile}"
       write (u, '(A)') "\end{document}"
    end subroutine cascade_set_write_graph_format
```

This is for screen output and debugging:

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_write
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_write (cascade_set, unit, active_only, complete_only)
      type(cascade_set_t), intent(in), target :: cascade_set
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: active_only, complete_only
      logical :: active, complete
      type(cascade_t), pointer :: cascade
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      active = .true.;  if (present (active_only))  active = active_only
      complete = .false.;  if (present (complete_only))  complete = complete_only
      write (u, "(A)") "Cascade set:"
      write (u, "(3x,A)", advance="no")  "Model:"
      if (associated (cascade_set%model)) then
         write (u, "(1x,A)") char (cascade_set%model%get_name ())
      else
         write (u, "(1x,A)") "[none]"
      end if
      write (u, "(3x,A)", advance="no")  "n_in/out/tot  ="
      write (u, "(3(1x,I7))")  &
           cascade_set%n_in, cascade_set%n_out, cascade_set%n_tot
      write (u, "(3x,A)", advance="no")  "depth_out/tot ="
      write (u, "(2(1x,I7))")  cascade_set%depth_out, cascade_set%depth_tot
      write (u, "(3x,A)", advance="no")  "mass thr(s/t) ="
      write (u, "(2(1x," // FMT_19 // "))")  &
           cascade_set%m_threshold_s, cascade_set%m_threshold_t
      write (u, "(3x,A)", advance="no")  "off shell     ="
      write (u, "(1x,I7)")  cascade_set%off_shell
      write (u, "(3x,A)", advance="no")  "keep_nonreson ="
      write (u, "(1x,L1)")  cascade_set%keep_nonresonant
      write (u, "(3x,A)", advance="no")  "n_groves      ="
      write (u, "(1x,I7)")  cascade_set%n_groves
      write (u, "(A)")
      write (u, "(A)") "Cascade list:"
      if (associated (cascade_set%first)) then
         cascade => cascade_set%first
         do while (associated (cascade))
            if (active .and. .not. cascade%active)  cycle
            if (complete .and. .not. cascade%complete)  cycle
            call cascade_write (cascade, unit)
            cascade => cascade%next
```

1506

```
      end do
   else
      write (u, "(A)") "[empty]"
   end if
   write (u, "(A)") "Hash array"
   write (u, "(3x,A)", advance="no")  "n_entries     ="
   write (u, "(1x,I7)")  cascade_set%n_entries
   write (u, "(3x,A)", advance="no")  "fill_ratio    ="
   write (u, "(1x," // FMT_12 // ")")  cascade_set%fill_ratio
   write (u, "(3x,A)", advance="no")  "n_entries_max ="
   write (u, "(1x,I7)")  cascade_set%n_entries_max
   write (u, "(3x,A)", advance="no")  "mask          ="
   write (u, "(1x,I0)")  cascade_set%mask
   do i = 0, ubound (cascade_set%entry, 1)
      if (allocated (cascade_set%entry(i)%key)) then
         write (u, "(1x,I7)") i
         call hash_entry_write (cascade_set%entry(i), u)
      end if
   end do
end subroutine cascade_set_write
```

11.7.7   Adding cascades

Add a deep copy of a cascade to the set. The copy has all content of the original,
but the pointers are nullified. We do not care whether insertion was successful
or not. The pointer argument, if present, is assigned to the input cascade, or to
the hash entry if it is already present.

The procedure is recursive: any daughter or mother entries are also deep-
copied and added to the cascade set before the current copy is added.

⟨*Cascades: procedures*⟩+≡
```
   recursive subroutine cascade_set_add_copy &
        (cascade_set, cascade_in, cascade_ptr)
     type(cascade_set_t), intent(inout), target :: cascade_set
     type(cascade_t), intent(in) :: cascade_in
     type(cascade_t), optional, pointer :: cascade_ptr
     type(cascade_t), pointer :: cascade
     logical :: ok
     allocate (cascade)
     cascade = cascade_in
     if (associated (cascade_in%daughter1))  call cascade_set_add_copy &
          (cascade_set, cascade_in%daughter1, cascade%daughter1)
     if (associated (cascade_in%daughter2))  call cascade_set_add_copy &
          (cascade_set, cascade_in%daughter2, cascade%daughter2)
     if (associated (cascade_in%mother))  call cascade_set_add_copy &
          (cascade_set, cascade_in%mother, cascade%mother)
     cascade%next => null ()
     call cascade_set_add (cascade_set, cascade, ok, cascade_ptr)
     if (.not. ok)  deallocate (cascade)
   end subroutine cascade_set_add_copy
```

Add a cascade to the set. This does not deep-copy. We first try to insert it in the hash array. If successful, add it to the list. Failure indicates that it is already present, and we drop it.

The hash key is built solely from the tree array, so neither particle codes nor resonances count, just topology.

Technically, hash and list receive only pointers, so the cascade can be considered as being in either of both. We treat it as part of the list.

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_add (cascade_set, cascade, ok, cascade_ptr)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out) :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  integer(i8), dimension(1) :: mold
  call cascade_set_hash_insert &
      (cascade_set, transfer (cascade%tree, mold), cascade, ok, cascade_ptr)
  if (ok)  call cascade_set_list_add (cascade_set, cascade)
end subroutine cascade_set_add
```

Add a new cascade to the list:

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_list_add (cascade_set, cascade)
  type(cascade_set_t), intent(inout) :: cascade_set
  type(cascade_t), intent(in), target :: cascade
  if (associated (cascade_set%last)) then
     cascade_set%last%next => cascade
  else
     cascade_set%first => cascade
  end if
  cascade_set%last => cascade
end subroutine cascade_set_list_add
```

Add a cascade entry to the hash array:

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_hash_insert &
      (cascade_set, key, cascade, ok, cascade_ptr)
  type(cascade_set_t), intent(inout), target :: cascade_set
  integer(i8), dimension(:), intent(in) :: key
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out) :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  integer(i32) :: h
  if (cascade_set%n_entries >= cascade_set%n_entries_max) &
      call cascade_set_hash_expand (cascade_set)
  h = hash (key)
  call cascade_set_hash_insert_rec &
      (cascade_set, h, h, key, cascade, ok, cascade_ptr)
end subroutine cascade_set_hash_insert
```

Double the hashtable size when necesssary:

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_hash_expand (cascade_set)
```

```
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(hash_entry_t), dimension(:), allocatable, target :: table_tmp
      type(cascade_p), pointer :: current
      integer :: i, s
      allocate (table_tmp (0:cascade_set%mask))
      table_tmp = cascade_set%entry
      deallocate (cascade_set%entry)
      s = 2 * size (table_tmp)
      cascade_set%n_entries = 0
      cascade_set%n_entries_max = s * cascade_set%fill_ratio
      cascade_set%mask = s - 1
      allocate (cascade_set%entry (0:cascade_set%mask))
      do i = 0, ubound (table_tmp, 1)
         current => table_tmp(i)%first
         do while (associated (current))
            call cascade_set_hash_insert_rec &
                 (cascade_set, table_tmp(i)%hashval, table_tmp(i)%hashval, &
                  table_tmp(i)%key, current%cascade)
            current => current%next
         end do
      end do
   end subroutine cascade_set_hash_expand
```

Insert the cascade at the bucket determined by the hash value. If the bucket is filled, check first for a collision (unequal keys). In that case, choose the following bucket and repeat. Otherwise, add the cascade to the bucket.

If the bucket is empty, record the hash value, allocate and store the key, and then add the cascade to the bucket.

If ok is present, before insertion we check whether the cascade is already stored, and return failure if yes.

⟨Cascades: procedures⟩+≡

```
   recursive subroutine cascade_set_hash_insert_rec &
        (cascade_set, h, hashval, key, cascade, ok, cascade_ptr)
     type(cascade_set_t), intent(inout) :: cascade_set
     integer(i32), intent(in) :: h, hashval
     integer(i8), dimension(:), intent(in) :: key
     type(cascade_t), intent(in), target :: cascade
     logical, intent(out), optional :: ok
     type(cascade_t), optional, pointer :: cascade_ptr
     integer(i32) :: i
     i = iand (h, cascade_set%mask)
     if (allocated (cascade_set%entry(i)%key)) then
        if (size (cascade_set%entry(i)%key) /= size (key)) then
           call cascade_set_hash_insert_rec &
                (cascade_set, h + 1, hashval, key, cascade, ok, cascade_ptr)
        else if (any (cascade_set%entry(i)%key /= key)) then
           call cascade_set_hash_insert_rec &
                (cascade_set, h + 1, hashval, key, cascade, ok, cascade_ptr)
        else
           call hash_entry_add_cascade_ptr &
                (cascade_set%entry(i), cascade, ok, cascade_ptr)
        end if
     else
```

```
            cascade_set%entry(i)%hashval = hashval
            allocate (cascade_set%entry(i)%key (size (key)))
            cascade_set%entry(i)%key = key
            call hash_entry_add_cascade_ptr &
                 (cascade_set%entry(i), cascade, ok, cascade_ptr)
            cascade_set%n_entries = cascade_set%n_entries + 1
         end if
      end subroutine cascade_set_hash_insert_rec
```

11.7.8   External particles

We want to initialize the cascade set with the outgoing particles. In case of
multiple processes, initial cascades are prepared for all of them. The hash array
check ensures that no particle appears more than once at the same place.

⟨*Cascades: interfaces*⟩+≡
```
      interface cascade_set_add_outgoing
         module procedure cascade_set_add_outgoing1
         module procedure cascade_set_add_outgoing2
      end interface
```

⟨*Cascades: procedures*⟩+≡
```
      subroutine cascade_set_add_outgoing2 (cascade_set, flv)
        type(cascade_set_t), intent(inout), target :: cascade_set
        type(flavor_t), dimension(:,:), intent(in) :: flv
        integer :: pos, prc, n_out, n_prc
        type(cascade_t), pointer :: cascade
        logical :: ok
        n_out = size (flv, dim=1)
        n_prc = size (flv, dim=2)
        do prc = 1, n_prc
           do pos = 1, n_out
              allocate (cascade)
              call cascade_init_outgoing &
                   (cascade, flv(pos,prc), pos, cascade_set%m_threshold_s)
              call cascade_set_add (cascade_set, cascade, ok)
              if (.not. ok) then
                 deallocate (cascade)
              end if
           end do
        end do
      end subroutine cascade_set_add_outgoing2

      subroutine cascade_set_add_outgoing1 (cascade_set, flv)
        type(cascade_set_t), intent(inout), target :: cascade_set
        type(flavor_t), dimension(:), intent(in) :: flv
        integer :: pos, n_out
        type(cascade_t), pointer :: cascade
        logical :: ok
        n_out = size (flv, dim=1)
        do pos = 1, n_out
           allocate (cascade)
           call cascade_init_outgoing &
```

```
                  (cascade, flv(pos), pos, cascade_set%m_threshold_s)
          call cascade_set_add (cascade_set, cascade, ok)
          if (.not. ok) then
              deallocate (cascade)
          end if
      end do
    end subroutine cascade_set_add_outgoing1
```

The incoming particles are added one at a time. Nevertheless, we may have
several processes which are looped over. At the first opportunity, we set the
pointer `first_t` in the cascade set which should point to the first t-channel
cascade.

Return the indices of the first and last cascade generated.

⟨*Cascades: interfaces*⟩+≡
```
    interface cascade_set_add_incoming
        module procedure cascade_set_add_incoming0
        module procedure cascade_set_add_incoming1
    end interface
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_add_incoming1 (cascade_set, n1, n2, pos, flv)
      type(cascade_set_t), intent(inout), target :: cascade_set
      integer, intent(out) :: n1, n2
      integer, intent(in) :: pos
      type(flavor_t), dimension(:), intent(in) :: flv
      integer :: prc, n_prc
      type(cascade_t), pointer :: cascade
      logical :: ok
      n1 = 0
      n2 = 0
      n_prc = size (flv)
      do prc = 1, n_prc
          allocate (cascade)
          call cascade_init_incoming &
              (cascade, flv(prc), pos, cascade_set%m_threshold_t)
          call cascade_set_add (cascade_set, cascade, ok)
          if (ok) then
              if (n1 == 0)  n1 = cascade%index
              n2 = cascade%index
              if (.not. associated (cascade_set%first_t)) then
                  cascade_set%first_t => cascade
              end if
          else
              deallocate (cascade)
          end if
      end do
    end subroutine cascade_set_add_incoming1

    subroutine cascade_set_add_incoming0 (cascade_set, n1, n2, pos, flv)
      type(cascade_set_t), intent(inout), target :: cascade_set
      integer, intent(out) :: n1, n2
      integer, intent(in) :: pos
      type(flavor_t), intent(in) :: flv
```

```
type(cascade_t), pointer :: cascade
logical :: ok
n1 = 0
n2 = 0
allocate (cascade)
call cascade_init_incoming &
     (cascade, flv, pos, cascade_set%m_threshold_t)
call cascade_set_add (cascade_set, cascade, ok)
if (ok) then
   if (n1 == 0)  n1 = cascade%index
   n2 = cascade%index
   if (.not. associated (cascade_set%first_t)) then
      cascade_set%first_t => cascade
   end if
else
   deallocate (cascade)
end if
end subroutine cascade_set_add_incoming0
```

11.7.9   Cascade combination I: flavor assignment

We have two disjunct cascades, now use the vertex table to determine the possible flavors of the combination cascade. For each possibility, try to generate a new cascade. The total cascade depth has to be one less than the limit, because this is reached by setting the keystone.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_match_pair (cascade_set, cascade1, cascade2, s_channel)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2
  logical, intent(in) :: s_channel
  integer, dimension(:), allocatable :: pdg3
  integer :: i, depth_max
  type(flavor_t) :: flv
  if (s_channel) then
     depth_max = cascade_set%depth_out
  else
     depth_max = cascade_set%depth_tot
  end if
  if (cascade1%depth + cascade2%depth < depth_max) then
     call model_match_vertex (cascade_set%model, &
          flavor_get_pdg (cascade1%flv), &
          flavor_get_pdg (cascade2%flv), &
          pdg3)
     do i = 1, size (pdg3)
        call flavor_init (flv, pdg3(i), cascade_set%model)
        if (s_channel) then
           call cascade_combine_s (cascade_set, cascade1, cascade2, flv)
        else
           call cascade_combine_t (cascade_set, cascade1, cascade2, flv)
        end if
     end do
     deallocate (pdg3)
```

```
            end if
        end subroutine cascade_match_pair
```

The triplet version takes a third cascade, and we check whether this triplet has a matching vertex in the database. If yes, we make a keystone cascade.

⟨*Cascades: procedures*⟩+≡
```
        subroutine cascade_match_triplet &
             (cascade_set, cascade1, cascade2, cascade3, s_channel)
          type(cascade_set_t), intent(inout), target :: cascade_set
          type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
          logical, intent(in) :: s_channel
          integer :: depth_max
          depth_max = cascade_set%depth_tot
          if (cascade1%depth + cascade2%depth + cascade3%depth == depth_max) then
             if (model_check_vertex (cascade_set%model, &
                   flavor_get_pdg (cascade1%flv), &
                   flavor_get_pdg (cascade2%flv), &
                   flavor_get_pdg (cascade3%flv))) then
                call cascade_combine_keystone &
                     (cascade_set, cascade1, cascade2, cascade3, s_channel)
             end if
          end if
        end subroutine cascade_match_triplet
```

11.7.10 Cascade combination II: kinematics setup and check

Having three matching flavors, we start constructing the combination cascade. We look at the mass hierarchies and determine whether the cascade is to be kept. In passing we set mapping modes, resonance properties and such.

If successful, the cascade is finalized. For a resonant cascade, we prepare in addition a copy without the resonance.

⟨*Cascades: procedures*⟩+≡
```
        subroutine cascade_combine_s (cascade_set, cascade1, cascade2, flv)
          type(cascade_set_t), intent(inout), target :: cascade_set
          type(cascade_t), intent(in), target :: cascade1, cascade2
          type(flavor_t), intent(in) :: flv
          type(cascade_t), pointer :: cascade3, cascade4
          logical :: keep
          keep = .false.
          allocate (cascade3)
          call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
          cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
          cascade3%flv = flavor_anti (flv)
          cascade3%pdg = abs (flavor_get_pdg (cascade3%flv))
          cascade3%is_vector = flavor_get_spin_type (flv) == VECTOR
          cascade3%m_min = cascade1%m_min + cascade2%m_min
          cascade3%m_rea = flavor_get_mass (flv)
          if (cascade3%m_rea > cascade_set%m_threshold_s) then
             cascade3%m_eff = cascade3%m_rea
          end if
          ! Potentially resonant cases [sqrts = m_rea for on-shell decay]
```

```
                  if (cascade3%m_rea > cascade3%m_min &
                      .and. cascade3%m_rea <= cascade_set%sqrts) then
                 if (flavor_get_width (flv) /= 0) then
                    if (cascade1%on_shell .or. cascade2%on_shell) then
                       keep = .true.
                       cascade3%mapping = S_CHANNEL
                       cascade3%resonant = .true.
                    end if
                 else
                    call warn_decay (flv)
                 end if
! Collinear and IR singular cases
else if (cascade3%m_rea < cascade_set%sqrts) then
   ! Massless splitting
   if (cascade1%m_eff == 0 .and. cascade2%m_eff == 0 &
         .and. cascade3%depth <= 3) then
      keep = .true.
      cascade3%log_enhanced = .true.
      if (cascade3%is_vector) then
         if (cascade1%is_vector .and. cascade2%is_vector) then
            cascade3%mapping = COLLINEAR   ! three-vector-vertex
         else
            cascade3%mapping = INFRARED    ! vector splitting into matter
         end if
      else
         if (cascade1%is_vector .or. cascade2%is_vector) then
            cascade3%mapping = COLLINEAR   ! vector radiation off matter
         else
            cascade3%mapping = INFRARED     ! scalar radiation/splitting
         end if
      end if
   ! IR radiation off massive particle
   else if (cascade3%m_eff > 0 .and. cascade1%m_eff > 0 &
         .and. cascade2%m_eff == 0 &
         .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
         .and. abs (cascade3%m_eff - cascade1%m_eff) &
                  < cascade_set%m_threshold_s) &
         then
      keep = .true.
      cascade3%log_enhanced = .true.
      cascade3%mapping = RADIATION
   else if (cascade3%m_eff > 0 .and. cascade2%m_eff > 0 &
         .and. cascade1%m_eff == 0 &
         .and. (cascade2%on_shell .or. cascade2%mapping == RADIATION) &
         .and. abs (cascade3%m_eff - cascade2%m_eff) &
              < cascade_set%m_threshold_s) &
         then
      keep = .true.
      cascade3%log_enhanced = .true.
      cascade3%mapping = RADIATION
   end if
end if
! Non-singular cases, including failed resonances
if (.not. keep) then
```

```fortran
      ! Two on-shell particles from a virtual mother
      if (cascade1%on_shell .or. cascade2%on_shell) then
         keep = .true.
         cascade3%m_eff = max (cascade3%m_min, &
                               cascade1%m_eff + cascade2%m_eff)
         if (cascade3%m_eff < cascade_set%m_threshold_s) then
            cascade3%m_eff = 0
         end if
      end if
   end if
   ! Complete and register the cascade (two in case of resonance)
   if (keep) then
      cascade3%on_shell = cascade3%resonant .or. cascade3%log_enhanced
      if (cascade3%resonant) then
         cascade3%pdg = abs (flavor_get_pdg (cascade3%flv))
         if (cascade_set%keep_nonresonant) then
            allocate (cascade4)
            cascade4 = cascade3
            cascade4%index = cascade_index ()
            cascade4%pdg = UNDEFINED
            cascade4%mapping = NO_MAPPING
            cascade4%resonant = .false.
            cascade4%on_shell = .false.
         end if
         cascade3%m_min = cascade3%m_rea
         call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
         if (cascade_set%keep_nonresonant) then
            call cascade_fusion (cascade_set, cascade1, cascade2, cascade4)
         end if
      else
         call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
      end if
   else
      deallocate (cascade3)
   end if
contains
   subroutine warn_decay (flv)
     type(flavor_t), intent(in) :: flv
     integer :: i
     integer, dimension(MAX_WARN_RESONANCE), save :: warned_code = 0
     LOOP_WARNED: do i = 1, MAX_WARN_RESONANCE
        if (warned_code(i) == 0) then
           warned_code(i) = flavor_get_pdg (flv)
           write (msg_buffer, "(A)") &
                & " Intermediate decay of zero-width particle " &
                & // char (flavor_get_name (flv)) &
                & // " may be possible."
           call msg_warning
           exit LOOP_WARNED
        else if (warned_code(i) == flavor_get_pdg (flv)) then
           exit LOOP_WARNED
        end if
     end do LOOP_WARNED
   end subroutine warn_decay
```

```
      end subroutine cascade_combine_s
```

⟨*Limits: public parameters*⟩+≡
```
      integer, parameter, public :: MAX_WARN_RESONANCE = 50
```

This is the t-channel version. `cascade1` is t-channel and contains the seed, `cascade2` is s-channel. We check for kinematically allowed beam decay (which is a fatal error), or massless splitting / soft radiation. The cascade is kept in all remaining cases and submitted for registration.

⟨*Cascades: procedures*⟩+≡
```
      subroutine cascade_combine_t (cascade_set, cascade1, cascade2, flv)
        type(cascade_set_t), intent(inout), target :: cascade_set
        type(cascade_t), intent(in), target :: cascade1, cascade2
        type(flavor_t), intent(in) :: flv
        type(cascade_t), pointer :: cascade3
        allocate (cascade3)
        call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
        cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
        cascade3%flv = flavor_anti (flv)
        cascade3%pdg = abs (flavor_get_pdg (cascade3%flv))
        cascade3%is_vector = flavor_get_spin_type (flv) == VECTOR
        if (cascade1%incoming) then
           cascade3%m_min = cascade2%m_min
        else
           cascade3%m_min = cascade1%m_min + cascade2%m_min
        end if
        cascade3%m_rea = flavor_get_mass (flv)
        if (cascade3%m_rea > cascade_set%m_threshold_t) then
           cascade3%m_eff = max (cascade3%m_rea, cascade2%m_eff)
        else if (cascade2%m_eff > cascade_set%m_threshold_t) then
           cascade3%m_eff = cascade2%m_eff
        else
           cascade3%m_eff = 0
        end if
        ! Allowed decay of beam particle
        if (cascade1%incoming &
             .and. cascade1%m_rea > cascade2%m_rea + cascade3%m_rea) then
           call beam_decay (cascade_set%fatal_beam_decay)
        ! Massless splitting
        else if (cascade1%m_eff == 0 &
             .and. cascade2%m_eff < cascade_set%m_threshold_t &
             .and. cascade3%m_eff == 0) then
           cascade3%mapping = U_CHANNEL
           cascade3%log_enhanced = .true.
        ! IR radiation off massive particle
        else if (cascade1%m_eff /= 0 .and. cascade2%m_eff == 0 &
             .and. cascade3%m_eff /= 0 &
             .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
             .and. abs (cascade1%m_eff - cascade3%m_eff) &
                 < cascade_set%m_threshold_t) &
             then
           cascade3%pdg = abs (flavor_get_pdg (flv))
           cascade3%log_enhanced = .true.
           cascade3%mapping = RADIATION
```

```
        end if
        cascade3%t_channel = .true.
        call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
      contains
        subroutine beam_decay (fatal_beam_decay)
          logical, intent(in) :: fatal_beam_decay
          write (msg_buffer, "(1x,A,1x,'->',1x,A,1x,A)") &
               char (flavor_get_name (cascade1%flv)), &
               char (flavor_get_name (cascade3%flv)), &
               char (flavor_get_name (cascade2%flv))
          call msg_message
          write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
               char (flavor_get_name (cascade1%flv)), cascade1%m_rea
          call msg_message
          write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
               char (flavor_get_name (cascade3%flv)), cascade3%m_rea
          call msg_message
          write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
               char (flavor_get_name (cascade2%flv)), cascade2%m_rea
          call msg_message
          if (fatal_beam_decay) then
             call msg_fatal (" Phase space: Initial beam particle can decay")
          else
             call msg_warning (" Phase space: Initial beam particle can decay")
          end if
        end subroutine beam_decay
    end subroutine cascade_combine_t
```

Here we complete a decay cascade. The third input is the single-particle cascade for the initial particle. There is no resonance or mapping assignment. The only condition for keeping the cascade is the mass sum of the final state, which must be less than the available energy.

Two modifications are necessary for scattering cascades: a pure s-channel diagram (cascade1 is the incoming particle) do not have a logarithmic mapping at top-level. And in a t-channel diagram, the last line exchanged is mapped t-channel, not u-channel. Finally, we can encounter the case of a $2 \rightarrow 1$ process, where cascade1 is incoming, and cascade2 is the outgoing particle. In all three cases we register a new cascade with the modified mapping.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_combine_keystone &
         (cascade_set, cascade1, cascade2, cascade3, s_channel)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
      logical, intent(in) :: s_channel
      type(cascade_t), pointer :: cascade4, cascade0
      logical :: keep, ok
      keep = .false.
      allocate (cascade4)
      call cascade_init &
           (cascade4, cascade1%depth + cascade2%depth + cascade3%depth)
      cascade4%complete = .true.
      if (s_channel) then
         cascade4%bincode = ior (cascade1%bincode, cascade2%bincode)
```

```
      else
         cascade4%bincode = cascade3%bincode
      end if
      cascade4%flv = cascade3%flv
      cascade4%pdg = cascade3%pdg
      cascade4%mapping = EXTERNAL_PRT
      cascade4%is_vector = cascade3%is_vector
      cascade4%m_min = cascade1%m_min + cascade2%m_min
      cascade4%m_rea = cascade3%m_rea
      cascade4%m_eff = cascade3%m_rea
      if (cascade4%m_min < cascade_set%sqrts) then
         keep = .true.
      end if
      if (keep) then
         if (cascade1%incoming .and. cascade2%log_enhanced) then
            allocate (cascade0)
            cascade0 = cascade2
            cascade0%next => null ()
            cascade0%index = cascade_index ()
            cascade0%mapping = NO_MAPPING
            cascade0%log_enhanced = .false.
            cascade0%n_log_enhanced = cascade0%n_log_enhanced - 1
            cascade0%tree_mapping(cascade0%depth) = NO_MAPPING
            call cascade_keystone &
                 (cascade_set, cascade1, cascade0, cascade3, cascade4, ok)
            if (ok) then
               call cascade_set_add (cascade_set, cascade0, ok)
            else
               deallocate (cascade0)
            end if
         else if (cascade1%t_channel .and. cascade1%mapping == U_CHANNEL) then
            allocate (cascade0)
            cascade0 = cascade1
            cascade0%next => null ()
            cascade0%index = cascade_index ()
            cascade0%mapping = T_CHANNEL
            cascade0%tree_mapping(cascade0%depth) = T_CHANNEL
            call cascade_keystone &
                 (cascade_set, cascade0, cascade2, cascade3, cascade4, ok)
            if (ok) then
               call cascade_set_add (cascade_set, cascade0, ok)
            else
               deallocate (cascade0)
            end if
         else if (cascade1%incoming .and. cascade2%depth == 1) then
            allocate (cascade0)
            cascade0 = cascade2
            cascade0%next => null ()
            cascade0%index = cascade_index ()
            cascade0%mapping = ON_SHELL
            cascade0%tree_mapping(cascade0%depth) = ON_SHELL
            call cascade_keystone &
                 (cascade_set, cascade1, cascade0, cascade3, cascade4, ok)
            if (ok) then
```

```
                 call cascade_set_add (cascade_set, cascade0, ok)
            else
               deallocate (cascade0)
            end if
         else
            call cascade_keystone &
                 (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
         end if
      else
         deallocate (cascade4)
      end if
   end subroutine cascade_combine_keystone
```

11.7.11 Cascade combination III: node connections and tree fusion

Here we assign global tree properties. If the allowed number of off-shell lines is exceeded, discard the new cascade. Otherwise, assign the trees, sort them, and assign connections. Finally, append the cascade to the list. This may fail (because in the hash array there is already an equivalent cascade). On failure, discard the cascade.

⟨*Cascades: procedures*⟩+≡
```
   subroutine cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
     type(cascade_set_t), intent(inout), target :: cascade_set
     type(cascade_t), intent(in), target :: cascade1, cascade2
     type(cascade_t), pointer :: cascade3
     integer :: i1, i2, i3, i4
     logical :: ok
     cascade3%internal = (cascade3%depth - 3) / 2
     if (cascade3%resonant) then
        cascade3%multiplicity = 1
        cascade3%n_resonances = &
             cascade1%n_resonances + cascade2%n_resonances + 1
     else
        cascade3%multiplicity = cascade1%multiplicity + cascade2%multiplicity
        cascade3%n_resonances = cascade1%n_resonances + cascade2%n_resonances
     end if
     if (cascade3%log_enhanced) then
        cascade3%n_log_enhanced = &
             cascade1%n_log_enhanced + cascade2%n_log_enhanced + 1
     else
        cascade3%n_log_enhanced = &
             cascade1%n_log_enhanced + cascade2%n_log_enhanced
     end if
     if (cascade3%resonant) then
        cascade3%n_off_shell = 0
     else if (cascade3%log_enhanced) then
        cascade3%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell
     else
        cascade3%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell + 1
     end if
     if (cascade3%t_channel) then
```

```
            cascade3%n_t_channel = cascade1%n_t_channel + 1
        end if
        if (cascade3%n_off_shell > cascade_set%off_shell) then
            deallocate (cascade3)
        else if (cascade3%n_t_channel > cascade_set%t_channel) then
            deallocate (cascade3)
        else
            i1 = cascade1%depth
            i2 = i1 + 1
            i3 = i1 + cascade2%depth
            i4 = cascade3%depth
            cascade3%tree(:i1) = cascade1%tree
            where (cascade1%tree_mapping > NO_MAPPING)
                cascade3%tree_pdg(:i1) = cascade1%tree_pdg
            elsewhere
                cascade3%tree_pdg(:i1) = UNDEFINED
            end where
            cascade3%tree_mapping(:i1) = cascade1%tree_mapping
            cascade3%tree_resonant(:i1) = cascade1%tree_resonant
            cascade3%tree(i2:i3) = cascade2%tree
            where (cascade2%tree_mapping > NO_MAPPING)
                cascade3%tree_pdg(i2:i3) = cascade2%tree_pdg
            elsewhere
                cascade3%tree_pdg(i2:i3) = UNDEFINED
            end where
            cascade3%tree_mapping(i2:i3) = cascade2%tree_mapping
            cascade3%tree_resonant(i2:i3) = cascade2%tree_resonant
            cascade3%tree(i4) = cascade3%bincode
            cascade3%tree_pdg(i4) = cascade3%pdg
            cascade3%tree_mapping(i4) = cascade3%mapping
            cascade3%tree_resonant(i4) = cascade3%resonant
            call tree_sort (cascade3%tree, &
                 cascade3%tree_pdg, cascade3%tree_mapping, cascade3%tree_resonant)
            cascade3%has_children = .true.
            cascade3%daughter1 => cascade1
            cascade3%daughter2 => cascade2
            call cascade_set_add (cascade_set, cascade3, ok)
            if (.not. ok)  deallocate (cascade3)
        end if
    end subroutine cascade_fusion
```

Here we combine a cascade pair with an incoming particle, i.e., we set a keystone. Otherwise, this is similar. On the first opportunity, we set the `first_k` pointer in the cascade set.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_keystone &
        (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
      type(cascade_t), pointer :: cascade4
      logical, intent(out) :: ok
      integer :: i1, i2, i3, i4
      cascade4%internal = (cascade4%depth - 3) / 2
```

```fortran
      cascade4%multiplicity = cascade1%multiplicity + cascade2%multiplicity
      cascade4%n_resonances = cascade1%n_resonances + cascade2%n_resonances
      cascade4%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell
      cascade4%n_log_enhanced = &
              cascade1%n_log_enhanced + cascade2%n_log_enhanced
      cascade4%n_t_channel = cascade1%n_t_channel + cascade2%n_t_channel
      if (cascade4%n_off_shell > cascade_set%off_shell) then
         deallocate (cascade4)
         ok = .false.
      else if (cascade4%n_t_channel > cascade_set%t_channel) then
         deallocate (cascade4)
         ok = .false.
      else
         i1 = cascade1%depth
         i2 = i1 + 1
         i3 = i1 + cascade2%depth
         i4 = cascade4%depth
         cascade4%tree(:i1) = cascade1%tree
         where (cascade1%tree_mapping > NO_MAPPING)
            cascade4%tree_pdg(:i1) = cascade1%tree_pdg
         elsewhere
            cascade4%tree_pdg(:i1) = UNDEFINED
         end where
         cascade4%tree_mapping(:i1) = cascade1%tree_mapping
         cascade4%tree_resonant(:i1) = cascade1%tree_resonant
         cascade4%tree(i2:i3) = cascade2%tree
         where (cascade2%tree_mapping > NO_MAPPING)
            cascade4%tree_pdg(i2:i3) = cascade2%tree_pdg
         elsewhere
            cascade4%tree_pdg(i2:i3) = UNDEFINED
         end where
         cascade4%tree_mapping(i2:i3) = cascade2%tree_mapping
         cascade4%tree_resonant(i2:i3) = cascade2%tree_resonant
         cascade4%tree(i4) = cascade4%bincode
         cascade4%tree_pdg(i4) = UNDEFINED
         cascade4%tree_mapping(i4) = cascade4%mapping
         cascade4%tree_resonant(i4) = .false.
         call tree_sort (cascade4%tree, &
              cascade4%tree_pdg, cascade4%tree_mapping, cascade4%tree_resonant)
         cascade4%has_children = .true.
         cascade4%daughter1 => cascade1
         cascade4%daughter2 => cascade2
         cascade4%mother => cascade3
         call cascade_set_add (cascade_set, cascade4, ok)
         if (ok) then
            if (.not. associated (cascade_set%first_k)) then
               cascade_set%first_k => cascade4
            end if
         else
            deallocate (cascade4)
         end if
      end if
  end subroutine cascade_keystone
```

Sort a tree (array of binary codes) and particle code array simultaneously, by ascending binary codes. A convenient method is to use the `maxloc` function iteratively, to find and remove the largest entry in the tree array one by one.

⟨*Cascades: procedures*⟩+≡

```
subroutine tree_sort (tree, pdg, mapping, resonant)
  integer(TC), dimension(:), intent(inout) :: tree
  integer, dimension(:), intent(inout) :: pdg, mapping
  logical, dimension(:), intent(inout) :: resonant
  integer(TC), dimension(size(tree)) :: tree_tmp
  integer, dimension(size(pdg)) :: pdg_tmp, mapping_tmp
  logical, dimension(size(resonant)) :: resonant_tmp
  integer, dimension(1) :: pos
  integer :: i
  tree_tmp = tree
  pdg_tmp = pdg
  mapping_tmp = mapping
  resonant_tmp = resonant
  do i = size(tree),1,-1
     pos = maxloc (tree_tmp)
     tree(i) = tree_tmp (pos(1))
     pdg(i) = pdg_tmp (pos(1))
     mapping(i) = mapping_tmp (pos(1))
     resonant(i) = resonant_tmp (pos(1))
     tree_tmp(pos(1)) = 0
  end do
end subroutine tree_sort
```

11.7.12 Cascade set generation

These procedures loop over cascades and build up the cascade set. After each iteration of the innermost loop, we set a breakpoint.

s-channel: We use a nested scan to combine all cascades with all other cascades.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_s (cascade_set)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), pointer :: cascade1, cascade2
  cascade1 => cascade_set%first
  LOOP1: do while (associated (cascade1))
     cascade2 => cascade_set%first
     LOOP2: do while (associated (cascade2))
        if (cascade2%index >= cascade1%index)  exit LOOP2
        if (cascade1 .disjunct. cascade2) then
           call cascade_match_pair (cascade_set, cascade1, cascade2, .true.)
        end if
        call terminate_now_if_signal ()
        cascade2 => cascade2%next
     end do LOOP2
     cascade1 => cascade1%next
  end do LOOP1
end subroutine cascade_set_generate_s
```

The t-channel cascades are directed and have a seed (one of the incoming particles) and a target (the other one). We loop over all possible seeds and targets. Inside this, we loop over all t-channel cascades (cascade1) and s-channel cascades (cascade2) and try to combine them.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_t (cascade_set, pos_seed, pos_target)
  type(cascade_set_t), intent(inout), target :: cascade_set
  integer, intent(in) :: pos_seed, pos_target
  type(cascade_t), pointer :: cascade_seed, cascade_target
  type(cascade_t), pointer :: cascade1, cascade2
  integer(TC) :: bc_seed, bc_target
  bc_seed = ibset (0_TC, pos_seed-1)
  bc_target = ibset (0_TC, pos_target-1)
  cascade_seed => cascade_set%first_t
  LOOP_SEED: do while (associated (cascade_seed))
     if (cascade_seed%bincode == bc_seed) then
        cascade_target => cascade_set%first_t
        LOOP_TARGET: do while (associated (cascade_target))
           if (cascade_target%bincode == bc_target) then
              cascade1 => cascade_set%first_t
              LOOP_T: do while (associated (cascade1))
                 if ((cascade1 .disjunct. cascade_target) &
                      .and. .not. (cascade1 .disjunct. cascade_seed)) then
                    cascade2 => cascade_set%first
                    LOOP_S: do while (associated (cascade2))
                       if ((cascade2 .disjunct. cascade_target) &
                            .and. (cascade2 .disjunct. cascade1)) then
                          call cascade_match_pair &
                               (cascade_set, cascade1, cascade2, .false.)
                       end if
                       call terminate_now_if_signal ()
                       cascade2 => cascade2%next
                    end do LOOP_S
                 end if
                 call terminate_now_if_signal ()
                 cascade1 => cascade1%next
              end do LOOP_T
           end if
           call terminate_now_if_signal ()
           cascade_target => cascade_target%next
        end do LOOP_TARGET
     end if
     call terminate_now_if_signal ()
     cascade_seed => cascade_seed%next
  end do LOOP_SEED
end subroutine cascade_set_generate_t
```

This part completes the phase space for decay processes. It is similar to s-channel cascade generation, but combines two cascade with the particular cascade of the incoming particle. This particular cascade is expected to be pointed at by `first_t`.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_decay (cascade_set)
```

```
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), pointer :: cascade1, cascade2
      type(cascade_t), pointer :: cascade_in
      cascade_in => cascade_set%first_t
      cascade1 => cascade_set%first
      do while (associated (cascade1))
         if (cascade1 .disjunct. cascade_in) then
            cascade2 => cascade1%next
            do while (associated (cascade2))
               if ((cascade2 .disjunct. cascade1) &
                    .and. (cascade2 .disjunct. cascade_in)) then
                  call cascade_match_triplet (cascade_set, &
                       cascade1, cascade2, cascade_in, .true.)
               end if
               call terminate_now_if_signal ()
               cascade2 => cascade2%next
            end do
         end if
         call terminate_now_if_signal ()
         cascade1 => cascade1%next
      end do
    end subroutine cascade_set_generate_decay
```

This part completes the phase space for scattering processes. We combine a t-channel cascade (containing the seed) with a s-channel cascade and the target.

⟨*Cascades: procedures*⟩+≡

```
    subroutine cascade_set_generate_scattering &
        (cascade_set, ns1, ns2, nt1, nt2, pos_seed, pos_target)
      type(cascade_set_t), intent(inout), target :: cascade_set
      integer, intent(in) :: pos_seed, pos_target
      integer, intent(in) :: ns1, ns2, nt1, nt2
      type(cascade_t), pointer :: cascade_seed, cascade_target
      type(cascade_t), pointer :: cascade1, cascade2
      integer(TC) :: bc_seed, bc_target
      bc_seed = ibset (0_TC, pos_seed-1)
      bc_target = ibset (0_TC, pos_target-1)
      cascade_seed => cascade_set%first_t
      LOOP_SEED: do while (associated (cascade_seed))
         if (cascade_seed%index < ns1) then
            cascade_seed => cascade_seed%next
            cycle LOOP_SEED
         else if (cascade_seed%index > ns2) then
            exit LOOP_SEED
         else if (cascade_seed%bincode == bc_seed) then
            cascade_target => cascade_set%first_t
            LOOP_TARGET: do while (associated (cascade_target))
               if (cascade_target%index < nt1) then
                  cascade_target => cascade_target%next
                  cycle LOOP_TARGET
               else if (cascade_target%index > nt2) then
                  exit LOOP_TARGET
               else if (cascade_target%bincode == bc_target) then
                  cascade1 => cascade_set%first_t
```

```
                        LOOP_T: do while (associated (cascade1))
                           if ((cascade1 .disjunct. cascade_target) &
                                .and. .not. (cascade1 .disjunct. cascade_seed)) then
                              cascade2 => cascade_set%first
                              LOOP_S: do while (associated (cascade2))
                                 if ((cascade2 .disjunct. cascade_target) &
                                      .and. (cascade2 .disjunct. cascade1)) then
                                    call cascade_match_triplet (cascade_set, &
                                         cascade1, cascade2, cascade_target, .false.)
                                 end if
                                 call terminate_now_if_signal ()
                                 cascade2 => cascade2%next
                              end do LOOP_S
                           end if
                           call terminate_now_if_signal ()
                           cascade1 => cascade1%next
                        end do LOOP_T
                     end if
                     call terminate_now_if_signal ()
                     cascade_target => cascade_target%next
                  end do LOOP_TARGET
               end if
               call terminate_now_if_signal ()
               cascade_seed => cascade_seed%next
            end do LOOP_SEED
       end subroutine cascade_set_generate_scattering
```

11.7.13 Groves

Before assigning groves, assign hashcodes to the resonance patterns, so they can
easily be compared.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_assign_resonance_hash (cascade_set)
      type(cascade_set_t), intent(inout) :: cascade_set
      type(cascade_t), pointer :: cascade
      cascade => cascade_set%first_k
      do while (associated (cascade))
         call cascade_assign_resonance_hash (cascade)
         cascade => cascade%next
      end do
    end subroutine cascade_set_assign_resonance_hash
```

After all cascades are recorded, we group the complete cascades in groves. A
grove consists of cascades with identical multiplicity, number of resonances, log-
enhanced, t-channel lines, and resonance flavors.

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_assign_groves (cascade_set)
      type(cascade_set_t), intent(inout), target :: cascade_set
      type(cascade_t), pointer :: cascade1, cascade2
      integer :: multiplicity
      integer :: n_resonances, n_log_enhanced, n_t_channel, n_off_shell
      integer :: res_hash
```

```
      integer :: grove
      grove = 0
      cascade1 => cascade_set%first_k
      do while (associated (cascade1))
         if (cascade1%active .and. cascade1%complete &
              .and. cascade1%grove == 0) then
            grove = grove + 1
            cascade1%grove = grove
            multiplicity = cascade1%multiplicity
            n_resonances = cascade1%n_resonances
            n_log_enhanced = cascade1%n_log_enhanced
            n_off_shell = cascade1%n_off_shell
            n_t_channel = cascade1%n_t_channel
            res_hash = cascade1%res_hash
            cascade2 => cascade1%next
            do while (associated (cascade2))
               if (cascade2%grove == 0) then
                  if (cascade2%multiplicity == multiplicity &
                       .and. cascade2%n_resonances == n_resonances &
                       .and. cascade2%n_log_enhanced == n_log_enhanced &
                       .and. cascade2%n_off_shell == n_off_shell &
                       .and. cascade2%n_t_channel == n_t_channel &
                       .and. cascade2%res_hash == res_hash) then
                     cascade2%grove = grove
                  end if
               end if
               call terminate_now_if_signal ()
               cascade2 => cascade2%next
            end do
         end if
         call terminate_now_if_signal ()
         cascade1 => cascade1%next
      end do
      cascade_set%n_groves = grove
    end subroutine cascade_set_assign_groves
```

11.7.14 Generate the phase space file

Generate a complete phase space configuration.

For each flavor assignment: First, all s-channel graphs that can be built up from the outgoing particles. Then we distinguish (1) decay, where we complete the s-channel graphs by connecting to the input line, and (2) scattering, where we now generate t-channel graphs by introducing an incoming particle, and complete this by connecting to the other incoming particle.

After all cascade sets have been generated, merge them into a common set. This eliminates redunancies between flavor assignments.

⟨*Cascades: public*⟩+≡
```
    public :: cascade_set_generate
```

⟨*Cascades: procedures*⟩+≡
```
    subroutine cascade_set_generate &
        (cascade_set, model, n_in, n_out, flv, phs_par, fatal_beam_decay)
```

```
type(cascade_set_t), intent(out) :: cascade_set
type(model_t), intent(in), target :: model
integer, intent(in) :: n_in, n_out
type(flavor_t), dimension(:,:), intent(in) :: flv
type(phs_parameters_t), intent(in) :: phs_par
logical, intent(in) :: fatal_beam_decay
type(cascade_set_t), dimension(:), allocatable :: cset
type(cascade_t), pointer :: cascade
integer :: i
if (phase_space_vanishes (phs_par%sqrts, n_in, flv))  return
call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
   fatal_beam_decay, flv)
allocate (cset (size (flv, 2)))
do i = 1, size (cset)
   call cascade_set_generate_single (cset(i), &
        model, n_in, n_out, flv(:,i), phs_par, fatal_beam_decay)
   cascade => cset(i)%first_k
   do while (associated (cascade))
      if (cascade%active .and. cascade%complete) then
         call cascade_set_add_copy (cascade_set, cascade)
      end if
      cascade => cascade%next
   end do
   call cascade_set_final (cset(i))
end do
cascade_set%first_k => cascade_set%first
call cascade_set_assign_resonance_hash (cascade_set)
call cascade_set_assign_groves (cascade_set)
end subroutine cascade_set_generate
```

This generates phase space for a single channel, without assigning groves.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_single (cascade_set, &
   model, n_in, n_out, flv, phs_par, fatal_beam_decay)
  type(cascade_set_t), intent(out) :: cascade_set
  type(model_t), intent(in), target :: model
  integer, intent(in) :: n_in, n_out
  type(flavor_t), dimension(:), intent(in) :: flv
  type(phs_parameters_t), intent(in) :: phs_par
  logical, intent(in) :: fatal_beam_decay
  integer :: n11, n12, n21, n22
  call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
     fatal_beam_decay)
  call cascade_set_add_outgoing (cascade_set, flv(n_in+1:))
  call cascade_set_generate_s (cascade_set)
  select case (n_in)
  case(1)
     call cascade_set_add_incoming &
          (cascade_set, n11, n12, n_out + 1, flv(1))
     call cascade_set_generate_decay (cascade_set)
  case(2)
     call cascade_set_add_incoming &
          (cascade_set, n11, n12, n_out + 1, flv(2))
```

```
      call cascade_set_add_incoming &
            (cascade_set, n21, n22, n_out + 2, flv(1))
      call cascade_set_generate_t (cascade_set, n_out + 1, n_out + 2)
      call cascade_set_generate_t (cascade_set, n_out + 2, n_out + 1)
      call cascade_set_generate_scattering &
            (cascade_set, n11, n12, n21, n22, n_out + 1, n_out + 2)
      call cascade_set_generate_scattering &
            (cascade_set, n21, n22, n11, n12, n_out + 2, n_out + 1)
    end select
  end subroutine cascade_set_generate_single
```

Sanity check: Before anything else is done, check if there could possibly be any phase space.

⟨*Cascades: procedures*⟩+≡
```
  function phase_space_vanishes (sqrts, n_in, flv) result (flag)
    logical :: flag
    real(default), intent(in) :: sqrts
    integer, intent(in) :: n_in
    type(flavor_t), dimension(:,:), intent(in) :: flv
    real(default), dimension(:,:), allocatable :: mass
    real(default), dimension(:), allocatable :: mass_in, mass_out
    integer :: n_prt, n_flv
    flag = .false.
    if (sqrts <= 0) then
       call msg_error ("Phase space vanishes (sqrts must be positive)")
       flag = .true.;   return
    end if
    n_prt = size (flv, 1)
    n_flv = size (flv, 2)
    allocate (mass (n_prt, n_flv), mass_in (n_flv), mass_out (n_flv))
    mass = flavor_get_mass (flv)
    mass_in = sum (mass(:n_in,:), 1)
    mass_out = sum (mass(n_in+1:,:), 1)
    if (any (mass_in > sqrts)) then
       call msg_error ("Mass sum of incoming particles " &
            // "is more than available energy")
       flag = .true.;   return
    end if
    if (any (mass_out > sqrts)) then
       call msg_error ("Mass sum of outgoing particles " &
            // "is more than available energy")
       flag = .true.;   return
    end if
  end function phase_space_vanishes
```

11.7.15   Unit tests

⟨*Cascades: public*⟩+≡
```
  public :: cascade_test
```

⟨*Cascades: procedures*⟩+≡
```
  subroutine cascade_test (u, results)
    integer, intent(in) :: u
```

```
      type(test_results_t), intent(inout) :: results
    ⟨Cascades: execute tests⟩
    end subroutine cascade_test
```

Checking the basic setup up of the phase space cascade parameterizations.

⟨*Cascades: execute tests*⟩≡
```
    call test (cascade_1, "cascade_1", &
         "check cascade setup", &
         u, results)
```

⟨*Cascades: tests*⟩≡
```
    subroutine cascade_1 (u)
      use os_interface
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t), dimension(5,2) :: flv
      type(cascade_set_t) :: cascade_set
      type(string_t) :: name, filename
      type(phs_parameters_t) :: phs_par
      name = "SM"
      filename = "SM.mdl"

      write (u, "(A)")  "* Test output: Cascades"
      write (u, "(A)")  "*   Purpose: test cascade phase space functions"
      write (u, "(A)")

      write (u, "(A)")  "* Initializing"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)

      call model_list%read_model (name, filename, os_data, model)
      call model_write (model, unit = u, verbose=.true.)

      call flavor_init (flv(1,1), 2, model)
      call flavor_init (flv(2,1),-2, model)
      call flavor_init (flv(3,1), 1, model)
      call flavor_init (flv(4,1),-1, model)
      call flavor_init (flv(5,1),21, model)
      call flavor_init (flv(1,2), 2, model)
      call flavor_init (flv(2,2),-2, model)
      call flavor_init (flv(3,2), 2, model)
      call flavor_init (flv(4,2),-2, model)
      call flavor_init (flv(5,2),21, model)
      phs_par%sqrts = 1000._default
      phs_par%off_shell = 2

      write (u, "(A)")
      write (u, "(A)")  "* Generating the cascades"
      write (u, "(A)")
```

```
      call cascade_set_generate (cascade_set, model, 2, 3, flv, phs_par,.true.)
      call cascade_set_write (cascade_set, u)
      call cascade_set_write_file_format (cascade_set, u)

      write (u, "(A)")  "* Cleanup"
      write (u, "(A)")

      call cascade_set_final (cascade_set)
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, *)
      write (u, "(A)")  "* Test output end: cascade_1"

    end subroutine cascade_1
```

11.8 WOOD phase space

This is the module that interfaces the `phs_forests` phase-space treatment and
the `cascades` module for generating phase-space channels. As an extension of
the `phs_base` abstract type, the phase-space configuration and instance imple-
ment the standard API.

(Currently, this is the only generic phase-space implementation of WHIZARD.
For trivial two-particle phase space, there is `phs_wood` as an alternative.)

⟨`phs_wood.f90`⟩≡
  ⟨*File header*⟩

```
  module phs_wood
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use unit_tests
    use os_interface
    use md5
    use constants !NODEP!
    use lorentz !NODEP!
    use variables
    use models
    use flavors
    use process_constants
    use sf_mappings
    use sf_base
    use phs_base
    use mappings
    use phs_forests
    use cascades
```

  ⟨*Standard module head*⟩

⟨*PHS wood: public*⟩

⟨*PHS wood: types*⟩

```
contains
```

⟨*PHS wood: procedures*⟩

⟨*PHS wood: tests*⟩

```
end module phs_wood
```

11.8.1 Configuration

⟨*PHS wood: public*⟩≡
```
    public :: phs_wood_config_t
```

⟨*PHS wood: types*⟩≡
```
    type, extends (phs_config_t) :: phs_wood_config_t
      character(32) :: md5sum_forest = ""
      integer :: io_unit = 0
      logical :: io_unit_keep_open = .false.
      logical :: use_equivalences = .false.
      logical :: fatal_beam_decay = .true.
      type(mapping_defaults_t) :: mapping_defaults
      type(phs_parameters_t) :: par
      type(string_t) :: run_id
      type(cascade_set_t), allocatable :: cascade_set
      type(phs_forest_t) :: forest
      type(os_data_t) :: os_data
     contains
     ⟨PHS wood: phs wood config: TBP⟩
    end type phs_wood_config_t
```

Finalizer. We should delete the cascade set and the forest subobject.

Also close the I/O unit, just in case. (We assume that `io_unit` is not standard input/output.)

⟨*PHS wood: phs wood config: TBP*⟩≡
```
    procedure :: final => phs_wood_config_final
```

⟨*PHS wood: procedures*⟩≡
```
    subroutine phs_wood_config_final (object)
      class(phs_wood_config_t), intent(inout) :: object
      logical :: opened
      if (object%io_unit /= 0) then
         inquire (unit = object%io_unit, opened = opened)
         if (opened)  close (object%io_unit)
      end if
      call object%clear_phase_space ()
      call phs_forest_final (object%forest)
    end subroutine phs_wood_config_final
```

Output. The contents of the PHS forest are not printed explicitly.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: write => phs_wood_config_write
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_write (object, unit)
      class(phs_wood_config_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)") &
           "Partonic phase-space configuration (phase-space forest):"
      call object%base_write (unit)
      write (u, "(1x,A)")    "Phase-space configuration parameters:"
      call phs_parameters_write (object%par, u)
      call object%mapping_defaults%write (u)
      write (u, "(3x,A,A,A)")  "Run ID: '", char (object%run_id), "'"
    end subroutine phs_wood_config_write
```

Print the PHS forest contents.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: write_forest => phs_wood_config_write_forest
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_write_forest (object, unit)
      class(phs_wood_config_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      call phs_forest_write (object%forest, u)
    end subroutine phs_wood_config_write_forest
```

Set the phase-space parameters that the configuration generator requests.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: set_parameters => phs_wood_config_set_parameters
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_set_parameters (phs_config, par)
      class(phs_wood_config_t), intent(inout) :: phs_config
      type(phs_parameters_t), intent(in) :: par
      phs_config%par = par
    end subroutine phs_wood_config_set_parameters
```

Enable the generation of channel equivalences (when calling `configure`).

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: enable_equivalences => phs_wood_config_enable_equivalences
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_enable_equivalences (phs_config)
      class(phs_wood_config_t), intent(inout) :: phs_config
      phs_config%use_equivalences = .true.
    end subroutine phs_wood_config_enable_equivalences
```

Set the phase-space mapping parameters that the configuration generator requests.g

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: set_mapping_defaults => phs_wood_config_set_mapping_defaults
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_set_mapping_defaults (phs_config, mapping_defaults)
      class(phs_wood_config_t), intent(inout) :: phs_config
      type(mapping_defaults_t), intent(in) :: mapping_defaults
      phs_config%mapping_defaults = mapping_defaults
    end subroutine phs_wood_config_set_mapping_defaults
```

Define the input stream for the phase-space file as an open logical unit. The unit must be connected.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: set_input => phs_wood_config_set_input
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_set_input (phs_config, unit)
      class(phs_wood_config_t), intent(inout) :: phs_config
      integer, intent(in) :: unit
      phs_config%io_unit = unit
      rewind (unit)
    end subroutine phs_wood_config_set_input
```

11.8.2 Phase-space generation

This subroutine generates a phase space configuration using the `cascades` module. Note that this may take time, and the `cascade_set` subobject may consume a large amount of memory.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: generate_phase_space => phs_wood_config_generate_phase_space
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_generate_phase_space (phs_config)
      class(phs_wood_config_t), intent(inout) :: phs_config
      integer :: off_shell, extra_off_shell
      call msg_message ("Phase space: generating configuration ...")
      off_shell = phs_config%par%off_shell
      allocate (phs_config%cascade_set)
      do extra_off_shell = 0, max (phs_config%n_tot - 3, 0)
         phs_config%par%off_shell = off_shell + extra_off_shell
         call cascade_set_generate (phs_config%cascade_set, &
             phs_config%model, phs_config%n_in, phs_config%n_out, &
             phs_config%flv, &
             phs_config%par, phs_config%fatal_beam_decay)
         if (cascade_set_is_valid (phs_config%cascade_set)) then
            exit
         else
            call msg_message ("Phase space: ... failed.  &
                &Increasing phs_off_shell ...")
         end if
      end do
```

```
        if (cascade_set_is_valid (phs_config%cascade_set)) then
           call msg_message ("Phase space: ... success.")
        else
           call msg_fatal ("Phase-space: generation failed")
        end if
     end subroutine phs_wood_config_generate_phase_space
```

Using the generated phase-space configuration, write an appropriate phase-space
file to the stored (or explicitly specified) I/O unit.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
     procedure :: write_phase_space => phs_wood_config_write_phase_space
```

⟨*PHS wood: procedures*⟩+≡
```
     subroutine phs_wood_config_write_phase_space (phs_config, &
           filename_vis, unit)
       class(phs_wood_config_t), intent(in) :: phs_config
       integer, intent(in), optional :: unit
       type(string_t), intent(in), optional :: filename_vis
       type(string_t) :: setenv_tex, setenv_mp, pipe, pipe_dvi
       integer :: u, unit_tex, unit_dev, status
       if (allocated (phs_config%cascade_set)) then
          if (present (unit)) then
             u = unit
          else
             u = phs_config%io_unit
          end if
          write (u, "(1x,A,A)") "process ", char (phs_config%id)
          write (u, "(A)")
          call cascade_set_write_process_bincode_format (phs_config%cascade_set, u)
          write (u, "(A)")
          write (u, "(3x,A,A,A32,A)") "md5sum_process    = ", &
               '"', phs_config%md5sum_process, '"'
          write (u, "(3x,A,A,A32,A)") "md5sum_model_par  = ", &
               '"', phs_config%md5sum_model_par, '"'
          write (u, "(3x,A,A,A32,A)") "md5sum_phs_config = ", &
               '"', phs_config%md5sum_phs_config, '"'
          call phs_parameters_write (phs_config%par, u)
          call cascade_set_write_file_format (phs_config%cascade_set, u)
          if (phs_config%vis_channels) then
             unit_tex = free_unit ()
             open (unit=unit_tex, file=char(filename_vis // ".tex"), &
                  action="write", status="replace")
             call cascade_set_write_graph_format (phs_config%cascade_set, &
                  filename_vis // "-graphs", phs_config%id, unit_tex)
             close (unit_tex)
             call msg_message ("Phase space: visualizing channels in file " &
                  // char(trim(filename_vis)) // "...")
             if (phs_config%os_data%event_analysis_ps) then
                BLOCK: do
                   unit_dev = free_unit ()
                   open (file = "/dev/null", unit = unit_dev, &
                        action = "write", iostat = status)
                   if (status /= 0) then
                      pipe = ""
```

```
                    pipe_dvi = ""
                 else
                    pipe = " > /dev/null"
                    pipe_dvi = " 2>/dev/null 1>/dev/null"
                 end if
                 close (unit_dev)
                 if (phs_config%os_data%whizard_texpath /= "") then
                    setenv_tex = "TEXINPUTS=" // &
                         phs_config%os_data%whizard_texpath // ":$TEXINPUTS "
                    setenv_mp = "MPINPUTS=" // &
                         phs_config%os_data%whizard_texpath // ":$MPINPUTS "
                 else
                    setenv_tex = ""
                    setenv_mp = ""
                 end if
                 call os_system_call (setenv_tex // &
                      phs_config%os_data%latex // " " // &
                      filename_vis // ".tex " // pipe, status)
                 if (status /= 0)  exit BLOCK
                 if (phs_config%os_data%mpost /= "") then
                    call os_system_call (setenv_mp // &
                         phs_config%os_data%mpost // " " // &
                         filename_vis // "-graphs.mp" // pipe, status)
                 else
                    call msg_fatal ("Could not use MetaPOST.")
                 end if
                 if (status /= 0)  exit BLOCK
                 call os_system_call (setenv_tex // &
                      phs_config%os_data%latex // " " // &
                      filename_vis // ".tex" // pipe, status)
                 if (status /= 0)  exit BLOCK
                 call os_system_call &
                      (phs_config%os_data%dvips // " -o " // filename_vis &
                      // ".ps " // filename_vis // ".dvi" // pipe_dvi, status)
                 if (status /= 0)  exit BLOCK
                 if (phs_config%os_data%event_analysis_pdf) then
                    call os_system_call (phs_config%os_data%ps2pdf // " " // &
                         filename_vis // ".ps", status)
                    if (status /= 0)  exit BLOCK
                 end if
                 exit BLOCK
              end do BLOCK
              if (status /= 0) then
                 call msg_error ("Unable to compile analysis output file")
              end if
           end if
        end if
     else
        call msg_fatal ("Phase-space configuration: &
              &no phase space object generated")
     end if
  end subroutine phs_wood_config_write_phase_space
```

Clear the phase-space configuration. This is useful since the object may become

really large.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: clear_phase_space => phs_wood_config_clear_phase_space
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_clear_phase_space (phs_config)
      class(phs_wood_config_t), intent(inout) :: phs_config
      if (allocated (phs_config%cascade_set)) then
         call cascade_set_final (phs_config%cascade_set)
         deallocate (phs_config%cascade_set)
      end if
    end subroutine phs_wood_config_clear_phase_space
```

11.8.3 Phase-space configuration

We read the phase-space configuration from the stored I/O unit. If this is not set, we assume that we have to generate a phase space configuration. When done, we open a scratch file and write the configuration.

If `rebuild` is set, we should trash any existing phase space file and build a new one. Otherwise, we try to use an old one, which we check for existence and integrity. If `ignore_mismatch` is set, we reuse an existing file even if it does not match the current setup.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: configure => phs_wood_config_configure
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_configure (phs_config, sqrts, &
         sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch)
      class(phs_wood_config_t), intent(inout) :: phs_config
      real(default), intent(in) :: sqrts
      logical, intent(in), optional :: sqrts_fixed
      logical, intent(in), optional :: cm_frame
      logical, intent(in), optional :: azimuthal_dependence
      logical, intent(in), optional :: rebuild
      logical, intent(in), optional :: ignore_mismatch
      type(string_t) :: filename, filename_vis
      logical :: variable_limits
      logical :: ok, exist, found, check, match, rebuild_phs
      integer :: g, c0, c1, n
      phs_config%sqrts = sqrts
      phs_config%par%sqrts = sqrts
      if (present (sqrts_fixed)) &
           phs_config%sqrts_fixed = sqrts_fixed
      if (present (cm_frame)) &
           phs_config%cm_frame = cm_frame
      if (present (azimuthal_dependence)) &
           phs_config%azimuthal_dependence = azimuthal_dependence
      if (present (rebuild)) then
         rebuild_phs = rebuild
      else
         rebuild_phs = .true.
      end if
      if (present (ignore_mismatch)) then
```

```
      check = .not. ignore_mismatch
      if (ignore_mismatch) &
           call msg_warning ("Reading phs file: MD5 sum check disabled")
   else
      check = .true.
   end if
   phs_config%md5sum_forest = ""
   call phs_config%compute_md5sum ()
   if (phs_config%io_unit == 0) then
      if (phs_config%run_id /= "") then
         filename = phs_config%id // "." // phs_config%run_id // ".phs"
         filename_vis = phs_config%id // "." // phs_config%run_id // "_phs"
      else
         filename = phs_config%id // ".phs"
         filename_vis = phs_config%id // "_phs"
      end if
      if (.not. rebuild_phs) then
         if (check) then
            call phs_config%read_phs_file (exist, found, match)
            rebuild_phs = .not. (exist .and. found .and. match)
         else
            call phs_config%read_phs_file (exist, found)
            rebuild_phs = .not. (exist .and. found)
         end if
      end if
      if (rebuild_phs) then
         call phs_config%generate_phase_space ()
         phs_config%io_unit = free_unit ()
         if (phs_config%id /= "") then
            call msg_message ("Phase space: writing configuration file '" &
                 // char (filename) // "'")
            open (phs_config%io_unit, file = char (filename), &
                 status = "replace", action = "readwrite")
         else
            open (phs_config%io_unit, status = "scratch", action = "readwrite")
         end if
         call phs_config%write_phase_space (filename_vis)
         rewind (phs_config%io_unit)
      else
         call msg_message ("Phase space: keeping configuration file '" &
              // char (filename) // "'")
      end if
   end if
   if (phs_config%io_unit == 0) then
      ok = .true.
   else
      call phs_forest_read (phs_config%forest, phs_config%io_unit, &
           phs_config%id, phs_config%n_in, phs_config%n_out, &
           phs_config%model, ok)
      if (.not. phs_config%io_unit_keep_open) then
         close (phs_config%io_unit)
         phs_config%io_unit = 0
      end if
   end if
```

```
      if (ok) then
         call phs_forest_set_flavors (phs_config%forest, phs_config%flv(:,1))
         variable_limits = .not. phs_config%cm_frame
         call phs_forest_set_parameters &
              (phs_config%forest, phs_config%mapping_defaults, variable_limits)
         call phs_forest_setup_prt_combinations (phs_config%forest)
         phs_config%n_channel = phs_forest_get_n_channels (phs_config%forest)
         phs_config%n_par = phs_forest_get_n_parameters (phs_config%forest)
         allocate (phs_config%channel (phs_config%n_channel))
         if (phs_config%use_equivalences) then
            call phs_forest_set_equivalences (phs_config%forest)
            call phs_forest_get_equivalences (phs_config%forest, &
                 phs_config%channel, phs_config%azimuthal_dependence)
            phs_config%provides_equivalences = .true.
         end if
         call phs_forest_set_s_mappings (phs_config%forest)
         call phs_config%record_on_shell ()
         if (phs_config%mapping_defaults%enable_s_mapping) then
            call phs_config%record_s_mappings ()
         end if
         allocate (phs_config%chain (phs_config%n_channel), source = 0)
         do g = 1, phs_forest_get_n_groves (phs_config%forest)
            call phs_forest_get_grove_bounds (phs_config%forest, g, c0, c1, n)
            phs_config%chain (c0:c1) = g
         end do
         phs_config%provides_chains = .true.
         call phs_config%compute_md5sum_forest ()
      else
         write (msg_buffer, "(A,A,A)") &
              "Phase space: process '", &
              char (phs_config%id), "' not found in configuration file"
         call msg_fatal ()
      end if
    end subroutine phs_wood_config_configure
```

Identify resonances which are marked by s-channel mappings for the whole phase space and report them to the channel array.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: record_s_mappings => phs_wood_config_record_s_mappings
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_record_s_mappings (phs_config)
      class(phs_wood_config_t), intent(inout) :: phs_config
      logical :: flag
      real(default) :: mass, width
      integer :: c
      do c = 1, phs_config%n_channel
         call phs_forest_get_s_mapping (phs_config%forest, c, flag, mass, width)
         if (flag) then
            if (mass == 0) then
               call msg_fatal ("Phase space: s-channel resonance " &
                    // " has zero mass")
            end if
            if (width == 0) then
```

```
              call msg_fatal ("Phase space: s-channel resonance " &
                    // " has zero width")
          end if
          call phs_config%channel(c)%set_resonant (mass, width)
       end if
    end do
  end subroutine phs_wood_config_record_s_mappings
```

Identify on-shell mappings for the whole phase space and report them to the channel array.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: record_on_shell => phs_wood_config_record_on_shell
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_record_on_shell (phs_config)
      class(phs_wood_config_t), intent(inout) :: phs_config
      logical :: flag
      real(default) :: mass
      integer :: c
      do c = 1, phs_config%n_channel
         call phs_forest_get_on_shell (phs_config%forest, c, flag, mass)
         if (flag) then
            call phs_config%channel(c)%set_on_shell (mass)
         end if
      end do
    end subroutine phs_wood_config_record_on_shell
```

The MD5 sum of the forest is computed in addition to the MD5 sum of the configuration. The reason is that the forest may depend on a user-provided external file. On the other hand, this MD5 sum encodes all information that is relevant for further processing. Therefore, the `get_md5sum` method returns this result, once it is available.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: compute_md5sum_forest => phs_wood_config_compute_md5sum_forest
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_compute_md5sum_forest (phs_config)
      class(phs_wood_config_t), intent(inout) :: phs_config
      integer :: u
      u = free_unit ()
      open (u, status = "scratch", action = "readwrite")
      call phs_config%write_forest (u)
      rewind (u)
      phs_config%md5sum_forest = md5sum (u)
      close (u)
    end subroutine phs_wood_config_compute_md5sum_forest
```

Return the most relevant MD5 sum. This overrides the method of the base type.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure :: get_md5sum => phs_wood_config_get_md5sum
```

⟨*PHS wood: procedures*⟩+≡
```
   function phs_wood_config_get_md5sum (phs_config) result (md5sum)
     class(phs_wood_config_t), intent(in) :: phs_config
     character(32) :: md5sum
     if (phs_config%md5sum_forest /= "") then
        md5sum = phs_config%md5sum_forest
     else
        md5sum = phs_config%md5sum_phs_config
     end if
   end function phs_wood_config_get_md5sum
```

Check whether a phase-space configuration for the current process exists. We
look for the phase-space file that should correspond to the current process. If
we find it, we check the MD5 sums stored in the file against the MD5 sums in
the current configuration (if required).

   If successful, read the PHS file.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
   procedure :: read_phs_file => phs_wood_read_phs_file
```

⟨*PHS wood: procedures*⟩+≡
```
   subroutine phs_wood_read_phs_file (phs_config, exist, found, match)
     class(phs_wood_config_t), intent(inout) :: phs_config
     logical, intent(out) :: exist
     logical, intent(out) :: found
     logical, intent(out), optional :: match
     type(string_t) :: filename
     integer :: u
     filename = phs_config%id // ".phs"
     inquire (file = char (filename), exist = exist)
     if (exist) then
        u = free_unit ()
        open (u, file = char (filename), action = "read", status = "old")
        call phs_forest_read (phs_config%forest, u, &
             phs_config%id, phs_config%n_in, phs_config%n_out, &
             phs_config%model, found, &
             phs_config%md5sum_process, &
             phs_config%md5sum_model_par, &
             phs_config%md5sum_phs_config, &
             match = match)
        close (u)
     else
        found = .false.
        if (present (match))  match = .false.
     end if
   end subroutine phs_wood_read_phs_file
```

Startup message, after configuration is complete.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
   procedure :: startup_message => phs_wood_config_startup_message
```

⟨*PHS wood: procedures*⟩+≡
```
   subroutine phs_wood_config_startup_message (phs_config, unit)
     class(phs_wood_config_t), intent(in) :: phs_config
```

```
        integer, intent(in), optional :: unit
        integer :: n_groves, n_eq
        n_groves = phs_forest_get_n_groves (phs_config%forest)
        n_eq = phs_forest_get_n_equivalences (phs_config%forest)
        call phs_config%base_startup_message (unit)
        if (phs_config%n_channel == 1) then
           write (msg_buffer, "(A,2(I0,A))") &
                "Phase space: found ", phs_config%n_channel, &
                " channel, collected in ", n_groves, &
                " grove."
        else if (n_groves == 1) then
           write (msg_buffer, "(A,2(I0,A))") &
                "Phase space: found ", phs_config%n_channel, &
                " channels, collected in ", n_groves, &
                " grove."
           else
           write (msg_buffer, "(A,2(I0,A))") &
                "Phase space: found ", phs_config%n_channel, &
                " channels, collected in ", &
                phs_forest_get_n_groves (phs_config%forest), &
                " groves."
        end if
        call msg_message (unit = unit)
        if (phs_config%use_equivalences) then
           if (n_eq == 1) then
              write (msg_buffer, "(A,I0,A)") &
                   "Phase space: Using ", n_eq, &
                   " equivalence between channels."
           else
              write (msg_buffer, "(A,I0,A)") &
                   "Phase space: Using ", n_eq, &
                   " equivalences between channels."
           end if
        else
           write (msg_buffer, "(A)") &
                "Phase space: no equivalences between channels used."
        end if
        call msg_message (unit = unit)
        write (msg_buffer, "(A,2(1x,I0,1x,A))") &
             "Phase space: wood"
        call msg_message (unit = unit)
      end subroutine phs_wood_config_startup_message
```

Allocate an instance: the actual phase-space object.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
    procedure, nopass :: allocate_instance => phs_wood_config_allocate_instance
```

⟨*PHS wood: procedures*⟩+≡
```
    subroutine phs_wood_config_allocate_instance (phs)
      class(phs_t), intent(inout), pointer :: phs
      allocate (phs_wood_t :: phs)
    end subroutine phs_wood_config_allocate_instance
```

11.8.4 Kinematics implementation

We generate $\cos\theta$ and ϕ uniformly, covering the solid angle.

⟨*PHS wood: public*⟩+≡
```
public :: phs_wood_t
```

⟨*PHS wood: types*⟩+≡
```
type, extends (phs_t) :: phs_wood_t
  real(default) :: sqrts = 0
  type(phs_forest_t) :: forest
 contains
 ⟨PHS wood: phs wood: TBP⟩
end type phs_wood_t
```

Output. The `verbose` setting is irrelevant, we just display the contents of the base object.

⟨*PHS wood: phs wood: TBP*⟩≡
```
procedure :: write => phs_wood_write
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_write (object, unit, verbose)
  class(phs_wood_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = output_unit (unit)
  call object%base_write (u)
end subroutine phs_wood_write
```

Write the forest separately.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: write_forest => phs_wood_write_forest
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_write_forest (object, unit)
  class(phs_wood_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  call phs_forest_write (object%forest, u)
end subroutine phs_wood_write_forest
```

Finalizer.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: final => phs_wood_final
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_final (object)
  class(phs_wood_t), intent(inout) :: object
  call phs_forest_final (object%forest)
end subroutine phs_wood_final
```

Initialization. We allocate arrays (`base_init`) and adjust the phase-space volume. The two-particle phase space volume is

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.55294034614 \times 10^{-5} \tag{11.64}$$

independent of the particle masses.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: init => phs_wood_init
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_init (phs, phs_config)
  class(phs_wood_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config
  call phs%base_init (phs_config)
  select type (phs_config)
  type is (phs_wood_config_t)
     phs%forest = phs_config%forest
  end select
end subroutine phs_wood_init
```

11.8.5 Evaluation

We compute the outgoing momenta from the incoming momenta and the input parameter set `r_in` in channel `r_in`. We also compute the `r` parameters and Jacobians `f` for all other channels.

We do *not* need to a apply a transformation from/to the c.m. frame, because in `phs_base` the momenta are already boosted to the c.m. frame before assigning them in the `phs` object, and inversely boosted when extracting them.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: evaluate_selected_channel => phs_wood_evaluate_selected_channel
procedure :: evaluate_other_channels => phs_wood_evaluate_other_channels
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_evaluate_selected_channel (phs, c_in, r_in)
  class(phs_wood_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  logical :: ok
  phs%q_defined = .false.
  if (phs%p_defined) then
     call phs_forest_set_prt_in (phs%forest, phs%p)
     phs%r(:,c_in) = r_in
     call phs_forest_evaluate_selected_channel (phs%forest, &
          c_in, phs%active_channel, &
          phs%sqrts_hat, phs%r, phs%f, phs%volume, ok)
     if (ok) then
        phs%q = phs_forest_get_momenta_out (phs%forest)
        phs%q_defined = .true.
     end if
  end if
end subroutine phs_wood_evaluate_selected_channel
```

```
      subroutine phs_wood_evaluate_other_channels (phs, c_in)
        class(phs_wood_t), intent(inout) :: phs
        integer, intent(in) :: c_in
        if (phs%q_defined) then
           call phs_forest_evaluate_other_channels (phs%forest, &
                c_in, phs%active_channel, &
                phs%sqrts_hat, phs%r, phs%f, combine=.true.)
           phs%r_defined = .true.
        end if
      end subroutine phs_wood_evaluate_other_channels
```

Inverse evaluation.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
      procedure :: inverse => phs_wood_inverse
```

⟨*PHS wood: procedures*⟩+≡
```
      subroutine phs_wood_inverse (phs)
        class(phs_wood_t), intent(inout) :: phs
        if (phs%p_defined .and. phs%q_defined) then
           call phs_forest_set_prt_in (phs%forest, phs%p)
           call phs_forest_set_prt_out (phs%forest, phs%q)
           call phs_forest_recover_channel (phs%forest, &
                1, &
                phs%sqrts_hat, phs%r, phs%f, phs%volume)
           call phs_forest_evaluate_other_channels (phs%forest, &
                1, phs%active_channel, &
                phs%sqrts_hat, phs%r, phs%f, combine=.false.)
           phs%r_defined = .true.
        end if
      end subroutine phs_wood_inverse
```

11.8.6   Unit tests

⟨*PHS wood: public*⟩+≡
```
      public :: phs_wood_test
```

⟨*PHS wood: tests*⟩≡
```
      subroutine phs_wood_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
```
⟨*PHS wood: execute tests*⟩
```
      end subroutine phs_wood_test
```

⟨*PHS wood: public*⟩+≡
```
      public :: phs_wood_vis_test
```

⟨*PHS wood: tests*⟩+≡
```
      subroutine phs_wood_vis_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
```
⟨*PHS wood: execute vis tests*⟩
```
      end subroutine phs_wood_vis_test
```

Phase-space configuration data

Construct and display a test phase-space configuration object. Also check the `azimuthal_dependence` flag.

This auxiliary routine writes a phase-space configuration file to unit `u_phs`.

⟨*PHS wood: public*⟩+≡
```
   public :: write_test_phs_file
```

⟨*PHS wood: tests*⟩+≡
```
   subroutine write_test_phs_file (u_phs, procname)
     integer, intent(in) :: u_phs
     type(string_t), intent(in), optional :: procname
     if (present (procname)) then
        write (u_phs, "(A,A)")  "process ", char (procname)
     else
        write (u_phs, "(A)")  "process testproc"
     end if
     write (u_phs, "(A,A)")  "   md5sum_process    = ", '""'
     write (u_phs, "(A,A)")  "   md5sum_model_par  = ", '""'
     write (u_phs, "(A,A)")  "   md5sum_phs_config = ", '""'
     write (u_phs, "(A)")  "   sqrts          = 1000"
     write (u_phs, "(A)")  "   m_threshold_s =   50"
     write (u_phs, "(A)")  "   m_threshold_t =  100"
     write (u_phs, "(A)")  "   off_shell = 2"
     write (u_phs, "(A)")  "   t_channel = 6"
     write (u_phs, "(A)")  "   keep_nonresonant = T"
     write (u_phs, "(A)")  "  grove #1"
     write (u_phs, "(A)")  "    tree 3"
   end subroutine write_test_phs_file
```

⟨*PHS wood: execute tests*⟩≡
```
   call test (phs_wood_1, "phs_wood_1", &
        "phase-space configuration", &
        u, results)
```

⟨*PHS wood: tests*⟩+≡
```
   subroutine phs_wood_1 (u)
     integer, intent(in) :: u
     type(os_data_t) :: os_data
     type(model_list_t) :: model_list
     type(model_t), pointer :: model
     type(process_constants_t) :: process_data
     class(phs_config_t), allocatable :: phs_data
     type(mapping_defaults_t) :: mapping_defaults
     real(default) :: sqrts
     integer :: u_phs, iostat
     character(32) :: buffer

     write (u, "(A)")  "* Test output: phs_wood_1"
     write (u, "(A)")  "*   Purpose: initialize and display &
          &phase-space configuration data"
     write (u, "(A)")

     call os_data_init (os_data)
     call syntax_model_file_init ()
```

1545

```
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_1"), process_data)

write (u, "(A)")  "* Create a scratch phase-space file"
write (u, "(A)")

u_phs = free_unit ()
open (u_phs, status = "scratch", action = "readwrite")
call write_test_phs_file (u_phs, var_str ("phs_wood_1"))
rewind (u_phs)
do
   read (u_phs, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do

write (u, "(A)")
write (u, "(A)")  "* Setup phase-space configuration object"
write (u, "(A)")

mapping_defaults%step_mapping = .false.

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_input (u_phs)
   call phs_data%set_mapping_defaults (mapping_defaults)
end select

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%write_forest (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

close (u_phs)
call phs_data%final ()
call model_list%final ()
```

```
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_wood_1"

    end subroutine phs_wood_1
```

Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

⟨*PHS wood: execute tests*⟩+≡
```
    call test (phs_wood_2, "phs_wood_2", &
         "phase-space evaluation", &
         u, results)
```

⟨*PHS wood: tests*⟩+≡
```
    subroutine phs_wood_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(flavor_t) :: flv
      type(process_constants_t) :: process_data
      real(default) :: sqrts, E
      class(phs_config_t), allocatable, target :: phs_data
      class(phs_t), pointer :: phs => null ()
      type(vector4_t), dimension(2) :: p, q
      integer :: u_phs

      write (u, "(A)")  "* Test output: phs_wood_2"
      write (u, "(A)")  "*   Purpose: test simple single-channel phase space"
      write (u, "(A)")

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)
      call flavor_init (flv, 25, model)

      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_wood_2"), process_data)
      u_phs = free_unit ()
      open (u_phs, status = "scratch", action = "readwrite")
      call write_test_phs_file (u_phs, var_str ("phs_wood_2"))
      rewind (u_phs)

      allocate (phs_wood_config_t :: phs_data)
      call phs_data%init (process_data, model)
      select type (phs_data)
      type is (phs_wood_config_t)
```

```
   call phs_data%set_input (u_phs)
end select

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flavor_get_mass (flv)**2), 3)
p(2) = vector4_moving (E,-sqrt (E**2 - flavor_get_mass (flv)**2), 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.125, 0.5"
write (u, "(A)")

call phs%evaluate_selected_channel (1, [0.125_default, 0.5_default])
call phs%evaluate_other_channels (1)
call phs%write (u)
write (u, "(A)")
select type (phs)
type is (phs_wood_t)
   call phs%write_forest (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
call phs%final ()
deallocate (phs)

call phs_data%allocate_instance (phs)
call phs%init (phs_data)
```

1548

```
call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call phs%write (u)
write (u, "(A)")
select type (phs)
type is (phs_wood_t)
   call phs%write_forest (u)
end select

call phs%final ()
deallocate (phs)

close (u_phs)
call phs_data%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: phs_wood_2"

end subroutine phs_wood_2
```

Phase-space generation

Generate phase space for a simple process.

⟨*PHS wood: execute tests*⟩+≡
```
call test (phs_wood_3, "phs_wood_3", &
     "phase-space generation", &
     u, results)
```

⟨*PHS wood: tests*⟩+≡
```
subroutine phs_wood_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(process_constants_t) :: process_data
  type(phs_parameters_t) :: phs_par
  class(phs_config_t), allocatable :: phs_data
  integer :: iostat
  character(80) :: buffer

  write (u, "(A)")  "* Test output: phs_wood_3"
  write (u, "(A)")  "*   Purpose: generate a phase-space configuration"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)
```

```
      call syntax_phs_forest_init ()

      write (u, "(A)")  "* Initialize a process and phase-space parameters"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_wood_3"), process_data)
      allocate (phs_wood_config_t :: phs_data)
      call phs_data%init (process_data, model)

      phs_par%sqrts = 1000
      select type (phs_data)
      type is (phs_wood_config_t)
         call phs_data%set_parameters (phs_par)
         phs_data%io_unit_keep_open = .true.
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Generate a scratch phase-space file"
      write (u, "(A)")

      call phs_data%configure (phs_par%sqrts)

      select type (phs_data)
      type is (phs_wood_config_t)
         rewind (phs_data%io_unit)
         do
            read (phs_data%io_unit, "(A)", iostat = iostat)  buffer
            if (iostat /= 0)  exit
            write (u, "(A)") trim (buffer)
         end do
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_wood_3"

    end subroutine phs_wood_3
```

Nontrivial process

Generate phase space for a $2 \rightarrow 3$ process.

⟨*PHS wood: execute tests*⟩+≡
```
    call test (phs_wood_4, "phs_wood_4", &
         "nontrivial process", &
         u, results)
```

⟨*PHS wood: tests*⟩+≡

```
subroutine phs_wood_4 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(process_constants_t) :: process_data
  type(phs_parameters_t) :: phs_par
  class(phs_config_t), allocatable, target :: phs_data
  integer :: iostat
  character(80) :: buffer
  class(phs_t), pointer :: phs => null ()
  real(default) :: E, pL
  type(vector4_t), dimension(2) :: p
  type(vector4_t), dimension(3) :: q

  write (u, "(A)")  "* Test output: phs_wood_4"
  write (u, "(A)")  "*   Purpose: generate a phase-space configuration"
  write (u, "(A)")

  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)

  call syntax_phs_forest_init ()

  write (u, "(A)")  "* Initialize a process and phase-space parameters"
  write (u, "(A)")

  process_data%id = "phs_wood_4"
  process_data%model_name = "Test"
  process_data%n_in = 2
  process_data%n_out = 3
  process_data%n_flv = 1
  allocate (process_data%flv_state (process_data%n_in + process_data%n_out, &
       process_data%n_flv))
  process_data%flv_state(:,1) = [25, 25, 25, 6, -6]

  allocate (phs_wood_config_t :: phs_data)
  call phs_data%init (process_data, model)

  phs_par%sqrts = 1000
  select type (phs_data)
  type is (phs_wood_config_t)
     call phs_data%set_parameters (phs_par)
     phs_data%io_unit_keep_open = .true.
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Generate a scratch phase-space file"
  write (u, "(A)")

  call phs_data%configure (phs_par%sqrts)
```

```
select type (phs_data)
type is (phs_wood_config_t)
   rewind (phs_data%io_unit)
   do
      read (phs_data%io_unit, "(A)", iostat = iostat)  buffer
      if (iostat /= 0)  exit
      write (u, "(A)") trim (buffer)
   end do
end select

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
   E = phs_data%sqrts / 2
   pL = sqrt (E**2 - flavor_get_mass (phs_data%flv(1,1))**2)
end select
p(1) = vector4_moving (E, pL, 3)
p(2) = vector4_moving (E, -pL, 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()

write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.1, 0.2, 0.3, 0.4, 0.5"
write (u, "(A)")

call phs%evaluate_selected_channel (1, &
     [0.1_default, 0.2_default, 0.3_default, 0.4_default, 0.5_default])
call phs%evaluate_other_channels (1)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
call phs%final ()
deallocate (phs)

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
```

```
    call phs%set_outgoing_momenta (q)

    call phs%inverse ()
    call phs%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call phs%final ()
    deallocate (phs)

    call phs_data%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_4"

  end subroutine phs_wood_4
```

Equivalences

Generate phase space for a simple process, including channel equivalences.

⟨*PHS wood: execute tests*⟩+≡
```
    call test (phs_wood_5, "phs_wood_5", &
        "equivalences", &
        u, results)
```

⟨*PHS wood: tests*⟩+≡
```
  subroutine phs_wood_5 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    type(process_constants_t) :: process_data
    type(phs_parameters_t) :: phs_par
    class(phs_config_t), allocatable :: phs_data

    write (u, "(A)")  "* Test output: phs_wood_5"
    write (u, "(A)")  "*   Purpose: generate a phase-space configuration"
    write (u, "(A)")

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list%read_model (var_str ("Test"), &
        var_str ("Test.mdl"), os_data, model)

    call syntax_phs_forest_init ()

    write (u, "(A)")  "* Initialize a process and phase-space parameters"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_wood_5"), process_data)
```

```
      allocate (phs_wood_config_t :: phs_data)
      call phs_data%init (process_data, model)

      phs_par%sqrts = 1000
      select type (phs_data)
      type is (phs_wood_config_t)
         call phs_data%set_parameters (phs_par)
         call phs_data%enable_equivalences ()
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Generate a scratch phase-space file"
      write (u, "(A)")

      call phs_data%configure (phs_par%sqrts)
      call phs_data%write (u)
      write (u, "(A)")

      select type (phs_data)
      type is (phs_wood_config_t)
         call phs_data%write_forest (u)
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_wood_5"

    end subroutine phs_wood_5
```

MD5 sum checks

Generate phase space for a simple process. Repeat this with and without parameter change.

⟨*PHS wood: execute tests*⟩+≡
```
    call test (phs_wood_6, "phs_wood_6", &
         "phase-space generation", &
         u, results)
```

⟨*PHS wood: tests*⟩+≡
```
    subroutine phs_wood_6 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_constants_t) :: process_data
      type(phs_parameters_t) :: phs_par
      class(phs_config_t), allocatable :: phs_data
```

```
logical :: exist, found, match
integer :: u_phs
character(*), parameter :: filename = "phs_wood_6_p.phs"
type(var_list_t), pointer :: var_list

write (u, "(A)")  "* Test output: phs_wood_6"
write (u, "(A)")  "*   Purpose: generate and check  phase-space file"
write (u, "(A)")

call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process and phase-space parameters"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_6"), process_data)
process_data%id = "phs_wood_6_p"
process_data%md5sum = "1234567890abcdef1234567890abcdef"
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)

phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
end select

write (u, "(A)")  "* Remove previous phs file, if any"
write (u, "(A)")

inquire (file = filename, exist = exist)
if (exist) then
   u_phs = free_unit ()
   open (u_phs, file = filename, action = "write")
   close (u_phs, status = "delete")
end if

write (u, "(A)")  "* Check phase-space file (should fail)"
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a phase-space file"
```

```
write (u, "(A)")

call phs_data%configure (phs_par%sqrts)

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

write (u, "(A)")
write (u, "(A)")  "* Check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify sqrts and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
phs_par%sqrts = 500
select type (phs_data)
type is (phs_wood_config_t)
```

```
      call phs_data%set_parameters (phs_par)
      phs_data%sqrts = phs_par%sqrts
      phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)   = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par) = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify process and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
process_data%md5sum = "7777777777777777777777777777777"
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)   = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par) = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select
```

```
write (u, "(A)")
write (u, "(A)")  "* Modify phs parameter and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
process_data%md5sum = "1234567890abcdef1234567890abcdef"
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
phs_par%off_shell = 17
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)   = '", &
    phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par) = '", &
    phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
    phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify model parameter and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
var_list => model_get_var_list_ptr (model)
call var_list_set_real (var_list, var_str ("ms"), 100._default, &
    is_known = .true.)
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
phs_par%off_shell = 1
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()
```

1558

```
      write (u, "(1x,A,A,A)")  "MD5 sum (process)   = '", &
           phs_data%md5sum_process, "'"
      write (u, "(1x,A,A,A)")  "MD5 sum (model par) = '", &
           phs_data%md5sum_model_par, "'"
      write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
           phs_data%md5sum_phs_config, "'"

      select type (phs_data)
      type is (phs_wood_config_t)
         call phs_data%read_phs_file (exist, found, match)
         write (u, "(1x,A,L1)")  "exist = ", exist
         write (u, "(1x,A,L1)")  "found = ", found
         write (u, "(1x,A,L1)")  "match = ", match
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call phs_data%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_wood_6"

    end subroutine phs_wood_6
```

⟨*PHS wood: execute vis tests*⟩≡
```
      call test (phs_wood_vis_1, "phs_wood_vis_1", &
           "visualizing phase space channels", &
           u, results)
```
⟨*PHS wood: tests*⟩+≡
```
      subroutine phs_wood_vis_1 (u)
        integer, intent(in) :: u
        type(os_data_t) :: os_data
        type(model_list_t) :: model_list
        type(model_t), pointer :: model
        type(process_constants_t) :: process_data
        class(phs_config_t), allocatable :: phs_data
        type(mapping_defaults_t) :: mapping_defaults
        type(string_t) :: vis_file, pdf_file, ps_file
        real(default) :: sqrts
        logical :: exist, exist_pdf, exist_ps
        integer :: u_phs, iostat, u_vis
        character(95) :: buffer

        write (u, "(A)")  "* Test output: phs_wood_vis_1"
        write (u, "(A)")  "*   Purpose: visualizing the &
             &phase-space configuration"
        write (u, "(A)")

        call os_data_init (os_data)
```

```
call syntax_model_file_init ()
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_vis_1"), process_data)

write (u, "(A)")  "* Create a scratch phase-space file"
write (u, "(A)")

u_phs = free_unit ()
open (u_phs, status = "scratch", action = "readwrite")
call write_test_phs_file (u_phs, var_str ("phs_wood_vis_1"))
rewind (u_phs)
do
   read (u_phs, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do

write (u, "(A)")
write (u, "(A)")  "* Setup phase-space configuration object"
write (u, "(A)")

mapping_defaults%step_mapping = .false.

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_input (u_phs)
   call phs_data%set_mapping_defaults (mapping_defaults)
   phs_data%os_data = os_data
   phs_data%io_unit = 0
   phs_data%io_unit_keep_open = .true.
   phs_data%vis_channels = .true.
end select

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%write_forest (u)
end select

vis_file = "phs_wood_vis_1_phs.tex"
```

```
   ps_file  = "phs_wood_vis_1_phs.ps"
   pdf_file = "phs_wood_vis_1_phs.pdf"
   inquire (file = char (vis_file), exist = exist)
   if (exist) then
      u_vis = free_unit ()
      open (u_vis, file = char (vis_file), action = "read", status = "old")
      iostat = 0
      do while (iostat == 0)
         read (u_vis, "(A)", iostat = iostat)  buffer
         if (iostat == 0)  write (u, "(A)")  trim (buffer)
      end do
      close (u_vis)
   else
      write (u, "(A)")  "[Visualize LaTeX file is missing]"
   end if
   inquire (file = char (ps_file), exist = exist_ps)
   if (exist_ps) then
      write (u, "(A)")  "[Visualize Postscript file exists and is nonempty]"
   else
      write (u, "(A)")  "[Visualize Postscript file is missing/non-regular]"
   end if
   inquire (file = char (pdf_file), exist = exist_pdf)
   if (exist_pdf) then
      write (u, "(A)")  "[Visualize PDF file exists and is nonempty]"
   else
      write (u, "(A)")  "[Visualize PDF file is missing/non-regular]"
   end if

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   close (u_phs)
   call phs_data%final ()
   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: phs_wood_vis_1"

 end subroutine phs_wood_vis_1
```

Chapter 12

Random Number Generation

The integration and event generation modules need a random-number generator. Here, we provide an abstract interface for a random-number generator and a concrete implementation.

Module `rng_base`: Abstract type for interfacing or implementing a r.n.g.

Module `rng_tao`: Implementation as an interface to the TAO random number generator which the VAMP package provides.

Module `selectors`: Select from a range of integers based on a set of weights and a random-number generator.

12.1 Generic Random-Number Generator

For all generator implementations, we define a `rng` type which represents the state of a random-number generator with the associated methods that produce a random number. Furthermore, we define a `rng_factory` type. An object of this type is capable of allocating a sequence of `rng` objects. These generator states should be, if possible, statistically independent, so they can be used in parallel in different places of the event-generation chain.

⟨`rng_base.f90`⟩≡
  ⟨*File header*⟩

  module rng_base

    use kinds !NODEP!
  ⟨*Use file utils*⟩
    use unit_tests

  ⟨*Standard module head*⟩

  ⟨*RNG base: public*⟩

  ⟨*RNG base: types*⟩

1562

⟨*RNG base: interfaces*⟩

⟨*RNG base: test types*⟩

```
contains
```

⟨*RNG base: tests*⟩

```
end module rng_base
```

12.1.1 Generator type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

⟨*RNG base: public*⟩≡
```
    public :: rng_t
```

⟨*RNG base: types*⟩≡
```
    type, abstract :: rng_t
     contains
    ⟨RNG base: rng: TBP⟩
    end type rng_t
```

The `init` method initializes the generator and sets a seed. We should implement the interface such that a single integer is sufficient for a seed.

The seed may be omitted. The behavior without seed is not defined, however.

⟨*RNG base: rng: TBP*⟩≡
```
    procedure (rng_init), deferred :: init
```

⟨*RNG base: interfaces*⟩≡
```
    abstract interface
       subroutine rng_init (rng, seed)
         import
         class(rng_t), intent(out) :: rng
         integer, intent(in), optional :: seed
       end subroutine rng_init
    end interface
```

The `final` method deallocates memory where necessary and allows for another call of `init` to reset the generator.

⟨*RNG base: rng: TBP*⟩+≡
```
    procedure (rng_final), deferred :: final
```

⟨*RNG base: interfaces*⟩+≡
```
    abstract interface
       subroutine rng_final (rng)
         import
         class(rng_t), intent(inout) :: rng
       end subroutine rng_final
    end interface
```

Output. We should, at least, identify the generator.

⟨*RNG base: rng: TBP*⟩+≡
```
procedure (rng_write), deferred :: write
```

⟨*RNG base: interfaces*⟩+≡
```
abstract interface
   subroutine rng_write (rng, unit, indent)
     import
     class(rng_t), intent(in) :: rng
     integer, intent(in), optional :: unit, indent
   end subroutine rng_write
end interface
```

These routines generate a single and an array of default-precision random numbers, respectively.

⟨*RNG base: rng: TBP*⟩+≡
```
generic :: generate => generate_single, generate_array
procedure (rng_generate_single), deferred :: generate_single
procedure (rng_generate_array), deferred :: generate_array
```

⟨*RNG base: interfaces*⟩+≡
```
abstract interface
   subroutine rng_generate_single (rng, x)
     import
     class(rng_t), intent(inout) :: rng
     real(default), intent(out) :: x
   end subroutine rng_generate_single
end interface

abstract interface
   subroutine rng_generate_array (rng, x)
     import
     class(rng_t), intent(inout) :: rng
     real(default), dimension(:), intent(out) :: x
   end subroutine rng_generate_array
end interface
```

12.1.2 RNG Factory

A factory object has a `make` method that allocates and initializes a new generator of appropriate type. It uses a 16-bit integer for initialization. For a real-life implementation, the factory should return a sequence of statistically independent generators, and for different seeds, the sequences should also be independent.

⟨*RNG base: public*⟩+≡
```
public :: rng_factory_t
```

⟨*RNG base: types*⟩+≡
```
type, abstract :: rng_factory_t
 contains
 ⟨RNG base: rng factory: TBP⟩
end type rng_factory_t
```

Output. Should be short, just report the seed and current state of the factory.

⟨RNG base: rng factory: TBP⟩≡
```
    procedure (rng_factory_write), deferred :: write
```

⟨RNG base: interfaces⟩+≡
```
    abstract interface
       subroutine rng_factory_write (object, unit)
         import
         class(rng_factory_t), intent(in) :: object
         integer, intent(in), optional :: unit
       end subroutine rng_factory_write
    end interface
```

Initialize. It should be possible to do this repeatedly, resetting the state. The default seed should be 0.

⟨RNG base: rng factory: TBP⟩+≡
```
    procedure (rng_factory_init), deferred :: init
```

⟨RNG base: interfaces⟩+≡
```
    abstract interface
       subroutine rng_factory_init (factory, seed)
         import
         class(rng_factory_t), intent(out) :: factory
         integer(i16), intent(in), optional :: seed
       end subroutine rng_factory_init
    end interface
```

Spawn a new generator.

⟨RNG base: rng factory: TBP⟩+≡
```
    procedure (rng_factory_make), deferred :: make
```

⟨RNG base: interfaces⟩+≡
```
    abstract interface
       subroutine rng_factory_make (factory, rng)
         import
         class(rng_factory_t), intent(inout) :: factory
         class(rng_t), intent(out), allocatable :: rng
       end subroutine rng_factory_make
    end interface
```

12.1.3   Unit tests

⟨RNG base: public⟩+≡
```
    public :: rng_base_test
```

⟨RNG base: tests⟩≡
```
    subroutine rng_base_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨RNG base: execute tests⟩
```
    end subroutine rng_base_test
```

1565

Test generator

The test 'mock' random generator generates a repeating series with the numbers
$0.1, 0.3, 0.5, 0.7, 0.9$. It has an integer stored as state. The integer must be one
of $1, 3, 5, 7, 9$.

⟨*RNG base: public*⟩+≡

```
    public :: rng_test_t
```

⟨*RNG base: test types*⟩≡

```
    type, extends (rng_t) :: rng_test_t
      integer :: state = 1
    contains
      procedure :: write => rng_test_write
      procedure :: init => rng_test_init
      procedure :: final => rng_test_final
      procedure :: generate_single => rng_test_generate_single
      procedure :: generate_array => rng_test_generate_array
    end type rng_test_t
```

Output. The state is a single number, so print it.

⟨*RNG base: tests*⟩+≡

```
    subroutine rng_test_write (rng, unit, indent)
      class(rng_test_t), intent(in) :: rng
      integer, intent(in), optional :: unit, indent
      integer :: u, ind
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call write_indent (u, ind)
      write (u, "(A,I0,A)")  "Random-number generator: &
           &test (state = ", rng%state, ")"
    end subroutine rng_test_write
```

The default seed is 1.

⟨*RNG base: tests*⟩+≡

```
    subroutine rng_test_init (rng, seed)
      class(rng_test_t), intent(out) :: rng
      integer, intent(in), optional :: seed
      if (present (seed))  rng%state = seed
    end subroutine rng_test_init
```

Nothing to finalize:

⟨*RNG base: tests*⟩+≡

```
    subroutine rng_test_final (rng)
      class(rng_test_t), intent(inout) :: rng
    end subroutine rng_test_final
```

Generate a single number and advance the state.

⟨*RNG base: tests*⟩+≡

```
    subroutine rng_test_generate_single (rng, x)
      class(rng_test_t), intent(inout) :: rng
      real(default), intent(out) :: x
      x = rng%state / 10._default
```

1566

```
      rng%state = mod (rng%state + 2, 10)
    end subroutine rng_test_generate_single
```

The array generator calls the single-item generator multiple times.

⟨*RNG base: tests*⟩+≡
```
    subroutine rng_test_generate_array (rng, x)
      class(rng_test_t), intent(inout) :: rng
      real(default), dimension(:), intent(out) :: x
      integer :: i
      do i = 1, size (x)
         call rng%generate (x(i))
      end do
    end subroutine rng_test_generate_array
```

Test Factory

This factory makes `rng_test_t` generators, initialized with integers 1, 3, 5, 7, 9 if given the input 0, 1, 2, 3, 4. The generators within one sequence are all identical, however.

⟨*RNG base: public*⟩+≡
```
    public :: rng_test_factory_t
```

⟨*RNG base: test types*⟩+≡
```
    type, extends (rng_factory_t) :: rng_test_factory_t
       integer :: seed = 1
     contains
     ⟨RNG base: rng test factory: TBP⟩
    end type rng_test_factory_t
```

Output.

⟨*RNG base: rng test factory: TBP*⟩≡
```
    procedure :: write => rng_test_factory_write
```

⟨*RNG base: tests*⟩+≡
```
    subroutine rng_test_factory_write (object, unit)
      class(rng_test_factory_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A,I0,A)")  "RNG factory: test (", object%seed, ")"
    end subroutine rng_test_factory_write
```

Initialize, translating the given seed.

⟨*RNG base: rng test factory: TBP*⟩+≡
```
    procedure :: init => rng_test_factory_init
```

⟨*RNG base: tests*⟩+≡
```
    subroutine rng_test_factory_init (factory, seed)
      class(rng_test_factory_t), intent(out) :: factory
      integer(i16), intent(in), optional :: seed
      if (present (seed))  factory%seed = mod (seed * 2 + 1, 10)
    end subroutine rng_test_factory_init
```

⟨*RNG base: rng test factory: TBP*⟩+≡

```
    procedure :: make => rng_test_factory_make
```

⟨*RNG base: tests*⟩+≡

```
    subroutine rng_test_factory_make (factory, rng)
      class(rng_test_factory_t), intent(inout) :: factory
      class(rng_t), intent(out), allocatable :: rng
      allocate (rng_test_t :: rng)
      select type (rng)
      type is (rng_test_t)
         call rng%init (int (factory%seed))
      end select
    end subroutine rng_test_factory_make
```

Generator check

Initialize the generator and draw random numbers.

⟨*RNG base: execute tests*⟩≡

```
    call test (rng_base_1, "rng_base_1", &
        "rng initialization and call", &
        u, results)
```

⟨*RNG base: tests*⟩+≡

```
    subroutine rng_base_1 (u)
      integer, intent(in) :: u
      class(rng_t), allocatable :: rng

      real(default) :: x
      real(default), dimension(2) :: x2

      write (u, "(A)")  "* Test output: rng_base_1"
      write (u, "(A)")  "*   Purpose: initialize and call a test random-number &
          &generator"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize generator"
      write (u, "(A)")

      allocate (rng_test_t :: rng)
      call rng%init (3)

      call rng%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Get random number"
      write (u, "(A)")

      call rng%generate (x)
      write (u, "(A,2(1x,F9.7))")  "x =", x

      write (u, "(A)")
      write (u, "(A)")  "* Get random number pair"
      write (u, "(A)")
```

1568

```
      call rng%generate (x2)
      write (u, "(A,2(1x,F9.7))")  "x =", x2

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call rng%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: rng_base_1"

   end subroutine rng_base_1
```

Factory check

Set up a factory and spawn generators.

⟨*RNG base: execute tests*⟩+≡
```
      call test (rng_base_2, "rng_base_2", &
           "rng factory", &
           u, results)
```

⟨*RNG base: tests*⟩+≡
```
   subroutine rng_base_2 (u)
     integer, intent(in) :: u
     type(rng_test_factory_t) :: rng_factory
     class(rng_t), allocatable :: rng

     write (u, "(A)")  "* Test output: rng_base_2"
     write (u, "(A)")  "*   Purpose: initialize and use a rng factory"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize factory"
     write (u, "(A)")

     call rng_factory%init ()
     call rng_factory%write (u)

     write (u, "(A)")
     write (u, "(A)")  "* Make a generator"
     write (u, "(A)")

     call rng_factory%make (rng)
     call rng%write (u)

     write (u, "(A)")
     write (u, "(A)")  "* Cleanup"

     call rng%final ()

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: rng_base_2"

   end subroutine rng_base_2
```

12.2 TAO Random-Number Generator

This module provides an implementation for the generic random-number generator. Actually, we interface the TAO random-number generator which is available via the VAMP package.

⟨rng_tao.f90⟩≡
  ⟨File header⟩

  module rng_tao

    use kinds !NODEP!
  ⟨Use file utils⟩
    use unit_tests
    use tao_random_numbers !NODEP!

    use rng_base

  ⟨Standard module head⟩

  ⟨RNG tao: public⟩

  ⟨RNG tao: types⟩

  contains

  ⟨RNG tao: procedures⟩

  ⟨RNG tao: tests⟩

  end module rng_tao

12.2.1 Generator type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

We keep the seed, in case we want to recover it later, and count the number of calls since seeding.

⟨RNG tao: public⟩≡
    public :: rng_tao_t

⟨RNG tao: types⟩≡
    type, extends (rng_t) :: rng_tao_t
       integer :: seed = 0
       integer :: n_calls = 0
       type(tao_random_state) :: state
     contains
     ⟨RNG tao: rng tao: TBP⟩
    end type rng_tao_t

1570

Output: Display seed and number of calls.

⟨RNG tao: rng tao: TBP⟩≡
```
procedure :: write => rng_tao_write
```

⟨RNG tao: procedures⟩≡
```
subroutine rng_tao_write (rng, unit, indent)
  class(rng_tao_t), intent(in) :: rng
  integer, intent(in), optional :: unit, indent
  integer :: u, ind
  u = output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  write (u, "(A)")  "TAO random-number generator:"
  call write_indent (u, ind)
  write (u, "(2x,A,I0)")  "seed  = ", rng%seed
  call write_indent (u, ind)
  write (u, "(2x,A,I0)")  "calls = ", rng%n_calls
end subroutine rng_tao_write
```

The `init` method initializes the generator and sets a seed. We should implement the interface such that a single integer is sufficient for a seed.

The seed may be omitted. The default seed is 0.

⟨RNG tao: rng tao: TBP⟩+≡
```
procedure :: init => rng_tao_init
```

⟨RNG tao: procedures⟩+≡
```
subroutine rng_tao_init (rng, seed)
  class(rng_tao_t), intent(out) :: rng
  integer, intent(in), optional :: seed
  if (present (seed))  rng%seed = seed
  call tao_random_create (rng%state, rng%seed)
end subroutine rng_tao_init
```

The `final` method deallocates memory where necessary and allows for another call of `init` to reset the generator.

⟨RNG tao: rng tao: TBP⟩+≡
```
procedure :: final => rng_tao_final
```

⟨RNG tao: procedures⟩+≡
```
subroutine rng_tao_final (rng)
  class(rng_tao_t), intent(inout) :: rng
  call tao_random_destroy (rng%state)
end subroutine rng_tao_final
```

These routines generate a single and an array of default-precision random numbers, respectively.

We have to convert from explicit double to abstract default precision. Under normal conditions, both are equivalent, however. Unless, someone decides to do single precision, there is always an interface for `tao_random_numbers`.

⟨RNG tao: rng tao: TBP⟩+≡
```
procedure :: generate_single => rng_tao_generate_single
procedure :: generate_array => rng_tao_generate_array
```

⟨*RNG tao: procedures*⟩+≡
```
    subroutine rng_tao_generate_single (rng, x)
      class(rng_tao_t), intent(inout) :: rng
      real(default), intent(out) :: x
      real(default) :: r
      call tao_random_number (rng%state, r)
      x = r
      rng%n_calls = rng%n_calls + 1
    end subroutine rng_tao_generate_single

    subroutine rng_tao_generate_array (rng, x)
      class(rng_tao_t), intent(inout) :: rng
      real(default), dimension(:), intent(out) :: x
      real(default) :: r
      integer :: i
      do i = 1, size (x)
         call tao_random_number (rng%state, r)
         x(i) = r
      end do
      rng%n_calls = rng%n_calls + size (x)
    end subroutine rng_tao_generate_array
```

Factory

This factory makes `rng_tao_t` generators, initialized with the seeds

$$s_i = s_0 * 2^{16} + i \tag{12.1}$$

where s_0 is the seed (a 16-bit integer) given to the factory object, and i is the index in the generated sequence of generators, starting with zero.

⟨*RNG tao: public*⟩+≡
```
    public :: rng_tao_factory_t
```

⟨*RNG tao: types*⟩+≡
```
    type, extends (rng_factory_t) :: rng_tao_factory_t
      integer(i16) :: s = 0
      integer(i16) :: i = 0
     contains
     ⟨RNG tao: rng tao factory: TBP⟩
    end type rng_tao_factory_t
```

Output.

⟨*RNG tao: rng tao factory: TBP*⟩≡
```
    procedure :: write => rng_tao_factory_write
```

⟨*RNG tao: procedures*⟩+≡
```
    subroutine rng_tao_factory_write (object, unit)
      class(rng_tao_factory_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A,2(I0,A))") &
           "RNG factory: tao (", object%s, ",", object%i, ")"
```

```
        end subroutine rng_tao_factory_write
```

Initialize, translating the given seed.

⟨*RNG tao: rng tao factory: TBP*⟩+≡
```
    procedure :: init => rng_tao_factory_init
```

⟨*RNG tao: procedures*⟩+≡
```
    subroutine rng_tao_factory_init (factory, seed)
      class(rng_tao_factory_t), intent(out) :: factory
      integer(i16), intent(in), optional :: seed
      if (present (seed))  factory%s = seed
    end subroutine rng_tao_factory_init
```

⟨*RNG tao: rng tao factory: TBP*⟩+≡
```
    procedure :: make => rng_tao_factory_make
```

⟨*RNG tao: procedures*⟩+≡
```
    subroutine rng_tao_factory_make (factory, rng)
      class(rng_tao_factory_t), intent(inout) :: factory
      class(rng_t), intent(out), allocatable :: rng
      allocate (rng_tao_t :: rng)
      select type (rng)
      type is (rng_tao_t)
         call rng%init (factory%s * 65536 + factory%i)
         factory%i = int (factory%i + 1, kind = i16)
      end select
    end subroutine rng_tao_factory_make
```

12.2.2   Unit tests

⟨*RNG tao: public*⟩+≡
```
    public :: rng_tao_test
```

⟨*RNG tao: tests*⟩≡
```
    subroutine rng_tao_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨RNG tao: execute tests⟩
    end subroutine rng_tao_test
```

Generator check

Initialize the generator and draw random numbers.

⟨*RNG tao: execute tests*⟩≡
```
    call test (rng_tao_1, "rng_tao_1", &
        "rng initialization and call", &
        u, results)
```

⟨*RNG tao: tests*⟩+≡
```
    subroutine rng_tao_1 (u)
      integer, intent(in) :: u
      class(rng_t), allocatable, target :: rng

      real(default) :: x
      real(default), dimension(2) :: x2

      write (u, "(A)")  "* Test output: rng_tao_1"
      write (u, "(A)")  "*   Purpose: initialize and call the TAO random-number &
          &generator"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize generator (default seed)"
      write (u, "(A)")

      allocate (rng_tao_t :: rng)
      call rng%init ()

      call rng%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Get random number"
      write (u, "(A)")

      call rng%generate (x)
      write (u, "(A,2(1x,F9.7))")  "x =", x

      write (u, "(A)")
      write (u, "(A)")  "* Get random number pair"
      write (u, "(A)")

      call rng%generate (x2)
      write (u, "(A,2(1x,F9.7))")  "x =", x2

      write (u, "(A)")
      call rng%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call rng%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: rng_tao_1"

    end subroutine rng_tao_1
```

Factory check

Set up a factory and spawn generators.

⟨*RNG tao: execute tests*⟩+≡

```
            call test (rng_tao_2, "rng_tao_2", &
                 "rng factory", &
                 u, results)
```

⟨*RNG tao: tests*⟩+≡

```
      subroutine rng_tao_2 (u)
        integer, intent(in) :: u
        type(rng_tao_factory_t) :: rng_factory
        class(rng_t), allocatable :: rng
        real(default) :: x

        write (u, "(A)")  "* Test output: rng_tao_2"
        write (u, "(A)")  "*   Purpose: initialize and use a rng factory"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize factory"
        write (u, "(A)")

        call rng_factory%init ()
        call rng_factory%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Make a generator"
        write (u, "(A)")

        call rng_factory%make (rng)
        call rng%write (u)
        call rng%generate (x)
        write (u, *)
        write (u, "(1x,A,F7.5)")  "x = ", x
        call rng%final ()
        deallocate (rng)

        write (u, "(A)")
        write (u, "(A)")  "* Repeat"
        write (u, "(A)")

        call rng_factory%make (rng)
        call rng%write (u)
        call rng%generate (x)
        write (u, *)
        write (u, "(1x,A,F7.5)")  "x = ", x
        call rng%final ()
        deallocate (rng)

        write (u, *)
        call rng_factory%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Initialize factory with different seed"
        write (u, "(A)")

        call rng_factory%init (1_i16)
        call rng_factory%write (u)
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Make a generator"
      write (u, "(A)")

      call rng_factory%make (rng)
      call rng%write (u)
      call rng%generate (x)
      write (u, *)
      write (u, "(1x,A,F7.5)")   "x = ", x
      call rng%final ()
      deallocate (rng)

      write (u, *)
      call rng_factory%write (u)

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: rng_tao_2"

    end subroutine rng_tao_2
```

12.3   Select from a weighted sample

⟨selectors.f90⟩≡
  ⟨*File header*⟩

```
  module selectors

    use kinds !NODEP!
```
  ⟨*Use file utils*⟩
```
    use unit_tests
    use diagnostics !NODEP!
    use rng_base
```

  ⟨*Standard module head*⟩

  ⟨*Selectors: public*⟩

  ⟨*Selectors: types*⟩

```
  contains
```

  ⟨*Selectors: procedures*⟩

  ⟨*Selectors: tests*⟩

```
  end module selectors
```

12.3.1   Selector type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

⟨Selectors: public⟩≡
```
    public :: selector_t
```

⟨Selectors: types⟩≡
```
    type :: selector_t
       integer, dimension(:), allocatable :: map
       real(default), dimension(:), allocatable :: weight
       real(default), dimension(:), allocatable :: acc
     contains
     ⟨Selectors: selector: TBP⟩
  end type selector_t
```

Display contents.

⟨Selectors: selector: TBP⟩≡
```
    procedure :: write => selector_write
```

⟨Selectors: procedures⟩≡
```
    subroutine selector_write (object, unit)
      class(selector_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "Selector: i, weight, acc. weight"
      if (allocated (object%weight)) then
         do i = 1, size (object%weight)
            write (u, "(3x,I0,1x,ES19.12,1x,ES19.12)") &
                 object%map(i), object%weight(i), object%acc(i)
         end do
      else
         write (u, "(3x,A)")  "[undefined]"
      end if
    end subroutine selector_write
```

We pack the input weight array such that zero-weight entries are removed. We also normalize it. This makes a `map` array for mapping the selected weight to the actual entry necessary.

⟨Selectors: selector: TBP⟩+≡
```
    procedure :: init => selector_init
```

⟨Selectors: procedures⟩+≡
```
    subroutine selector_init (selector, weight)
      class(selector_t), intent(out) :: selector
      real(default), dimension(:), intent(in) :: weight
      real(default) :: s
      integer :: n, i
      logical, dimension(:), allocatable :: mask
      if (size (weight) == 0) &
           call msg_bug ("Selector init: zero-size weight array")
      if (any (weight < 0)) &
           call msg_bug ("Selector init: negative weight")
      s = sum (weight)
      if (s == 0) &
           call msg_bug ("Selector init: all weights are zero")
      allocate (mask (size (weight)), &
```

```
         source = weight /= 0)
    n = count (mask)
    allocate (selector%map (n), &
         source = pack ([(i, i = 1, size (weight))], mask))
    allocate (selector%weight (n), &
         source = pack (weight / s, mask))
    allocate (selector%acc (n))
    selector%acc(1) = selector%weight(1)
    do i = 2, n - 1
        selector%acc(i) = selector%acc(i-1) + selector%weight(i)
    end do
    selector%acc(n) = 1
end subroutine selector_init
```

Select an entry based upon the number x, which should be a uniformly distributed random number between 0 and 1.

⟨*Selectors: selector: TBP*⟩+≡
```
    procedure :: select => selector_select
```

⟨*Selectors: procedures*⟩+≡
```
    function selector_select (selector, x) result (n)
      class(selector_t), intent(in) :: selector
      real(default), intent(in) :: x
      integer :: n
      integer :: i
      if (x < 0 .or. x > 1) &
          call msg_bug ("Selector: random number out of range")
      do i = 1, size (selector%acc)
         if (x <= selector%acc(i))  exit
      end do
      n = selector%map(i)
    end function selector_select
```

Use the provided random-number generator to select an entry. (Unless there is only one entry.)

⟨*Selectors: selector: TBP*⟩+≡
```
    procedure :: generate => selector_generate
```

⟨*Selectors: procedures*⟩+≡
```
    subroutine selector_generate (selector, rng, n)
      class(selector_t), intent(in) :: selector
      class(rng_t), intent(inout) :: rng
      integer, intent(out) :: n
      real(default) :: x
      select case (size (selector%acc))
      case (1);  n = 1
      case default
         call rng%generate (x)
         n = selector%select (x)
      end select
    end subroutine selector_generate
```

Determine the normalized weight for a selected entry. We use a linear search for the inverse lookup, assuming that efficiency is not an issue for this function.

⟨*Selectors: selector: TBP*⟩+≡
```
    procedure :: get_weight => selector_get_weight
```

⟨*Selectors: procedures*⟩+≡
```
    function selector_get_weight (selector, n) result (weight)
      class(selector_t), intent(in) :: selector
      integer, intent(in) :: n
      real(default) :: weight
      integer :: i
      do i = 1, size (selector%weight)
         if (selector%map(i) == n) then
            weight = selector%weight(i)
            return
         end if
      end do
      weight = 0
    end function selector_get_weight
```

12.3.2 Unit tests

⟨*Selectors: public*⟩+≡
```
    public :: selectors_test
```

⟨*Selectors: tests*⟩≡
```
    subroutine selectors_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Selectors: execute tests*⟩
```
    end subroutine selectors_test
```

Basic check

Initialize the selector and draw random numbers.

⟨*Selectors: execute tests*⟩≡
```
    call test (selectors_1, "selectors_1", &
         "rng initialization and call", &
         u, results)
```

⟨*Selectors: tests*⟩+≡
```
    subroutine selectors_1 (u)
      integer, intent(in) :: u
      type(selector_t) :: selector
      class(rng_t), allocatable, target :: rng
      integer :: i, n

      write (u, "(A)")  "* Test output: selectors_1"
      write (u, "(A)")  "*   Purpose: initialize a selector and test it"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize selector"
```

```
    write (u, "(A)")

    call selector%init &
         ([2._default, 3.5_default, 0._default, &
          2._default, 0.5_default, 2._default])
    call selector%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Select numbers using predictable test generator"
    write (u, "(A)")

    allocate (rng_test_t :: rng)
    call rng%init (1)

    do i = 1, 5
       call selector%generate (rng, n)
       write (u, "(1x,I0)")  n
    end do

    write (u, "(A)")
    write (u, "(A)")  "* Select numbers using real input number"
    write (u, "(A)")

    write (u, "(1x,A,I0)")  "select(0.00) = ", selector%select (0._default)
    write (u, "(1x,A,I0)")  "select(0.77) = ", selector%select (0.77_default)
    write (u, "(1x,A,I0)")  "select(1.00) = ", selector%select (1._default)

    write (u, "(A)")
    write (u, "(A)")  "* Get weight"
    write (u, "(A)")

    write (u, "(1x,A,ES19.12)")  "weight(2) =", selector%get_weight(2)
    write (u, "(1x,A,ES19.12)")  "weight(3) =", selector%get_weight(3)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: selectors_1"

  end subroutine selectors_1
```

Chapter 13

Multi-Channel Integration

In this chapter we provide a generic module for handling multi-channel phase space and concrete implementations.

Module `mci_base`**:** The abstract types and their methods. It provides a test integrator that is referenced in later unit tests.

Module `mci_midpoint`**:** A simple integrator that uses the midpoint rule to sample the integrand uniformly over the unit hypercube. There is only one integration channel, so this can be matched only to single-channel phase space.

Module `mci_vamp`**:** Interface for the VAMP package.

13.1   Generic Integrator

This module provides a multi-channel integrator (MCI) base type, a corresponding configuration type, and methods for integration and event generation.

⟨`mci_base.f90`⟩≡
  ⟨*File header*⟩

  module mci_base

    use kinds !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: FMT_14, FMT_17 !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use cputime

    use phs_base
    use rng_base

  ⟨*Standard module head*⟩

  ⟨*MCI base: public*⟩

⟨*MCI base: types*⟩

⟨*MCI base: interfaces*⟩

⟨*MCI base: test types*⟩

```
contains
```

⟨*MCI base: procedures*⟩

⟨*MCI base: tests*⟩

```
end module mci_base
```

13.1.1 MCI: integrator

The MCI object contains the methods for integration and event generation. For the actual work and data storage, it spawns an MCI instance object.

The base object contains the number of integration dimensions and the number of channels as configuration data. Further configuration data are stored in the concrete extensions.

The MCI sum contains all relevant information about the integrand. It can be used for comparing the current configuration against a previous one. If they match, we can skip an actual integration. (Implemented only for the VAMP version.)

There is a random-number generator (its state with associated methods) available as `rng`. It may or may not be used for integration. It will be used for event generation.

⟨*MCI base: public*⟩≡
```
    public :: mci_t
```

⟨*MCI base: types*⟩≡
```
    type, abstract :: mci_t
       integer :: n_dim = 0
       integer :: n_channel = 0
       integer :: n_chain = 0
       integer, dimension(:), allocatable :: chain
       real(default), dimension(:), allocatable :: chain_weights
       character(32) :: md5sum = ""
       logical :: integral_known = .false.
       logical :: error_known = .false.
       logical :: efficiency_known = .false.
       real(default) :: integral = 0
       real(default) :: error = 0
       real(default) :: efficiency = 0
       logical :: use_timer = .false.
       type(timer_t) :: timer
       class(rng_t), allocatable :: rng
     contains
     ⟨MCI base: mci: TBP⟩
   end type mci_t
```

Finalizer: the random-number generator may need one.

⟨MCI base: mci: TBP⟩≡

```
procedure :: base_final => mci_final
procedure (mci_final), deferred :: final
```

⟨MCI base: procedures⟩≡

```
subroutine mci_final (object)
  class(mci_t), intent(inout) :: object
  if (allocated (object%rng))  call object%rng%final ()
end subroutine mci_final
```

Output: basic and extended output.

⟨MCI base: mci: TBP⟩+≡

```
procedure :: base_write => mci_write
procedure (mci_write), deferred :: write
```

⟨MCI base: procedures⟩+≡

```
subroutine mci_write (object, unit, pacify, md5sum_version)
  class(mci_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  logical, intent(in), optional :: md5sum_version
  logical :: md5sum_ver
  integer :: u, i, j
  character(len=7) :: fmt
  call pac_fmt (fmt, FMT_17, FMT_14, pacify)
  u = output_unit (unit)
  md5sum_ver = .false.
  if (present (md5sum_version))  md5sum_ver = md5sum_version
  if (object%use_timer .and. .not. md5sum_ver) then
     write (u, "(2x)", advance="no")
     call object%timer%write (u)
  end if
  if (object%integral_known) then
     write (u, "(3x,A," // fmt // ")") &
          "Integral             = ", object%integral
  end if
  if (object%error_known) then
     write (u, "(3x,A," // fmt // ")") &
          "Error                = ", object%error
  end if
  if (object%efficiency_known) then
     write (u, "(3x,A," // fmt // ")")  &
          "Efficiency           = ", object%efficiency
  end if
  write (u, "(3x,A,I0)")  "Number of channels   = ", object%n_channel
  write (u, "(3x,A,I0)")  "Number of dimensions = ", object%n_dim
  if (object%n_chain > 0) then
     write (u, "(3x,A,I0)")  "Number of chains     = ", object%n_chain
     write (u, "(3x,A)")  "Chains:"
     do i = 1, object%n_chain
        write (u, "(5x,I0,':')", advance = "no")  i
        do j = 1, object%n_channel
           if (object%chain(j) == i) &
```

1583

```
                write (u, "(1x,I0)", advance = "no")  j
           end do
           write (u, "(A)")
        end do
      end if
    end subroutine mci_write
```

Print an informative message when starting integration.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure (mci_startup_message), deferred :: startup_message
    procedure :: base_startup_message => mci_startup_message
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_startup_message (mci, unit, n_calls)
      class(mci_t), intent(in) :: mci
      integer, intent(in), optional :: unit, n_calls
      if (mci%n_chain > 0) then
         write (msg_buffer, "(A,3(1x,I0,1x,A))") &
               "Integrator:", mci%n_chain, "chains,", &
               mci%n_channel, "channels,", &
               mci%n_dim, "dimensions"
      else
         write (msg_buffer, "(A,3(1x,I0,1x,A))") &
               "Integrator:", &
               mci%n_channel, "channels,", &
               mci%n_dim, "dimensions"
      end if
      call msg_message (unit = unit)
    end subroutine mci_startup_message
```

Record the index of the MCI object within a process. For multi-component processes with more than one integrator, the integrator should know about its own index, so file names can be unique, etc. The default implementation does nothing, however.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: record_index => mci_record_index
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_record_index (mci, i_mci)
      class(mci_t), intent(inout) :: mci
      integer, intent(in) :: i_mci
    end subroutine mci_record_index
```

There is no Initializer for the abstract type, but a generic setter for the number of channels and dimensions. We make two aliases available, to be able to override it.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: set_dimensions => mci_set_dimensions
    procedure :: base_set_dimensions => mci_set_dimensions
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_set_dimensions (mci, n_dim, n_channel)
      class(mci_t), intent(inout) :: mci
```

1584

```
      integer, intent(in) :: n_dim
      integer, intent(in) :: n_channel
      mci%n_dim = n_dim
      mci%n_channel = n_channel
    end subroutine mci_set_dimensions
```

Declare particular dimensions as flat. This information can be used to simplify integration. When generating events, the flat dimensions should be sampled with uniform and uncorrelated distribution. It depends on the integrator what to do with that information.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure (mci_declare_flat_dimensions), deferred :: declare_flat_dimensions
```

⟨*MCI base: interfaces*⟩≡
```
    abstract interface
      subroutine mci_declare_flat_dimensions (mci, dim_flat)
        import
        class(mci_t), intent(inout) :: mci
        integer, dimension(:), intent(in) :: dim_flat
      end subroutine mci_declare_flat_dimensions
    end interface
```

Declare particular channels as equivalent, possibly allowing for permutations or reflections of dimensions. We use the information stored in the `phs_channel_t` object array that the phase-space module provides.

(We do not test this here, deferring the unit test to the `mci_vamp` implementation where we actually use this feature.)

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure (mci_declare_equivalences), deferred :: declare_equivalences
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
      subroutine mci_declare_equivalences (mci, channel, dim_offset)
        import
        class(mci_t), intent(inout) :: mci
        type(phs_channel_t), dimension(:), intent(in) :: channel
        integer, intent(in) :: dim_offset
      end subroutine mci_declare_equivalences
    end interface
```

Declare particular channels as chained together. The implementation may use this array for keeping their weights equal to each other, etc.

The chain array is an array sized by the number of channels. For each channel, there is an integer entry that indicates the corresponding chains. The total number of chains is the maximum value of this entry.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: declare_chains => mci_declare_chains
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_declare_chains (mci, chain)
      class(mci_t), intent(inout) :: mci
      integer, dimension(:), intent(in) :: chain
      allocate (mci%chain (size (chain)))
```

```
      mci%n_chain = maxval (chain)
      allocate (mci%chain_weights (mci%n_chain), source = 0._default)
      mci%chain = chain
    end subroutine mci_declare_chains
```

Collect channel weights according to chains and store them in the `chain_weights`
for output. We sum up the weights for all channels that share the same `chain`
index and store the results in the `chain_weights` array.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: collect_chain_weights => mci_collect_chain_weights
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_collect_chain_weights (mci, weight)
      class(mci_t), intent(inout) :: mci
      real(default), dimension(:), intent(in) :: weight
      integer :: i, c
      if (allocated (mci%chain)) then
         mci%chain_weights = 0
         do i = 1, size (mci%chain)
            c = mci%chain(i)
            mci%chain_weights(c) = mci%chain_weights(c) + weight(i)
         end do
      end if
    end subroutine mci_collect_chain_weights
```

Check if there are chains.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: has_chains => mci_has_chains
```

⟨*MCI base: procedures*⟩+≡
```
    function mci_has_chains (mci) result (flag)
      class(mci_t), intent(in) :: mci
      logical :: flag
      flag = allocated (mci%chain)
    end function mci_has_chains
```

Output of the chain weights, kept separate from the main `write` method.
[The formatting will work as long as the number of chains is less than 10^{10}. . .]

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: write_chain_weights => mci_write_chain_weights
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_write_chain_weights (mci, unit)
      class(mci_t), intent(in) :: mci
      integer, intent(in), optional :: unit
      integer :: u, i, n, n_digits
      character(4) :: ifmt
      u = output_unit (unit)
      if (allocated (mci%chain_weights)) then
         write (u, "(1x,A)")  "Weights of channel chains (groves):"
         n_digits = 0
         n = size (mci%chain_weights)
         do while (n > 0)
```

1586

```
        n = n / 10
        n_digits = n_digits + 1
      end do
      write (ifmt, "(A1,I1)") "I", n_digits
      do i = 1, size (mci%chain_weights)
         write (u, "(3x," // ifmt // ",F13.10)")  i, mci%chain_weights(i)
      end do
    end if
  end subroutine mci_write_chain_weights
```

Set the MD5 sum, independent of initialization.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: set_md5sum => mci_set_md5sum
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_set_md5sum (mci, md5sum)
      class(mci_t), intent(inout) :: mci
      character(32), intent(in) :: md5sum
      mci%md5sum = md5sum
    end subroutine mci_set_md5sum
```

Initialize a new integration pass. This is not necessarily meaningful, so we provide an empty base method. The `mci_vamp` implementation overrides this.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: add_pass => mci_add_pass
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_add_pass (mci, adapt_grids, adapt_weights, final)
      class(mci_t), intent(inout) :: mci
      logical, intent(in), optional :: adapt_grids
      logical, intent(in), optional :: adapt_weights
      logical, intent(in), optional :: final
    end subroutine mci_add_pass
```

Allocate an instance with matching type. This must be deferred.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure (mci_allocate_instance), deferred :: allocate_instance
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
       subroutine mci_allocate_instance (mci, mci_instance)
         import
         class(mci_t), intent(in) :: mci
         class(mci_instance_t), intent(out), pointer :: mci_instance
       end subroutine mci_allocate_instance
    end interface
```

Import a random-number generator. We transfer the allocation of an existing generator state into the object. The generator state may already be initialized, or we can reset it by its `init` method.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: import_rng => mci_import_rng
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_import_rng (mci, rng)
  class(mci_t), intent(inout) :: mci
  class(rng_t), intent(inout), allocatable :: rng
  call move_alloc (rng, mci%rng)
end subroutine mci_import_rng
```

Activate or deactivate the timer.
⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: set_timer => mci_set_timer
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_set_timer (mci, active)
  class(mci_t), intent(inout) :: mci
  logical, intent(in) :: active
  mci%use_timer = active
end subroutine mci_set_timer
```

Start and stop signal for the timer, if active. The elapsed time can then be retrieved from the MCI record.
⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: start_timer => mci_start_timer
procedure :: stop_timer => mci_stop_timer
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_start_timer (mci)
  class(mci_t), intent(inout) :: mci
  if (mci%use_timer)  call mci%timer%start ()
end subroutine mci_start_timer

subroutine mci_stop_timer (mci)
  class(mci_t), intent(inout) :: mci
  if (mci%use_timer)  call mci%timer%stop ()
end subroutine mci_stop_timer
```

Sampler test. Evaluate the sampler a given number of times. Results are discarded, so we don't need the MCI instance which would record them.

The evaluation channel is iterated, and the x parameters are randomly chosen.
⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: sampler_test => mci_sampler_test
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_sampler_test (mci, sampler, n_calls)
  class(mci_t), intent(inout) :: mci
  class(mci_sampler_t), intent(inout), target :: sampler
  integer, intent(in) :: n_calls
  real(default), dimension(:), allocatable :: x_in, f
  real(default), dimension(:,:), allocatable :: x_out
  real(default) :: val
  integer :: i, c
  allocate (x_in (mci%n_dim))
  allocate (f (mci%n_channel))
```

1588

```
      allocate (x_out (mci%n_dim, mci%n_channel))
      do i = 1, n_calls
         c = mod (i, mci%n_channel) + 1
         call mci%rng%generate_array (x_in)
         call sampler%evaluate (c, x_in, val, x_out, f)
      end do
   end subroutine mci_sampler_test
```

Integrate: this depends on the implementation. We foresee a pacify flag to take care of small numerical noise on different platforms.

⟨*MCI base: mci: TBP*⟩+≡
```
   procedure (mci_integrate), deferred :: integrate
```

⟨*MCI base: interfaces*⟩+≡
```
   abstract interface
      subroutine mci_integrate (mci, instance, sampler, &
           n_it, n_calls, results, pacify)
        import
        class(mci_t), intent(inout) :: mci
        class(mci_instance_t), intent(inout) :: instance
        class(mci_sampler_t), intent(inout) :: sampler
        integer, intent(in) :: n_it
        integer, intent(in) :: n_calls
        logical, intent(in), optional :: pacify
        class(mci_results_t), intent(inout), optional :: results
      end subroutine mci_integrate
   end interface
```

Event generation. Depending on the implementation, event generation may or may not require a previous integration pass.

Instead of a black-box `simulate` method, we require an initializer, a finalizer, and procedures for generating a single event. This allows us to interface simulation event by event from the outside, and it facilitates the further processing of an event after successful generation. For integration, this is not necessary.

The initializer has `intent(inout)` for the `mci` passed object. The reason is that the initializer can read integration results and grids from file, where the results can modify the `mci` record.

⟨*MCI base: mci: TBP*⟩+≡
```
   procedure (mci_prepare_simulation), deferred :: prepare_simulation
```

⟨*MCI base: interfaces*⟩+≡
```
   abstract interface
      subroutine mci_prepare_simulation (mci)
        import
        class(mci_t), intent(inout) :: mci
      end subroutine mci_prepare_simulation
   end interface
```

The generated event will reside in in the `instance` object (overall results and weight) and in the `sampler` object (detailed data). In the real application, we can subsequently call methods of the `sampler` in order to further process the generated event.

The `target` attributes are required by the VAMP implementation, which uses pointers to refer to the instance and sampler objects from within the integration function.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure (mci_generate), deferred :: generate_weighted_event
    procedure (mci_generate), deferred :: generate_unweighted_event
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
       subroutine mci_generate (mci, instance, sampler)
         import
         class(mci_t), intent(inout) :: mci
         class(mci_instance_t), intent(inout), target :: instance
         class(mci_sampler_t), intent(inout), target :: sampler
       end subroutine mci_generate
    end interface
```

This is analogous, but we rebuild the event from the information stored in `state` instead of generating it.

Note: currently unused outside of tests, might be deleted later.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure (mci_rebuild), deferred :: rebuild_event
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
       subroutine mci_rebuild (mci, instance, sampler, state)
         import
         class(mci_t), intent(inout) :: mci
         class(mci_instance_t), intent(inout) :: instance
         class(mci_sampler_t), intent(inout) :: sampler
         class(mci_state_t), intent(in) :: state
       end subroutine mci_rebuild
    end interface
```

Return the value of the integral, error, efficiency, and time per call.

⟨*MCI base: mci: TBP*⟩+≡
```
    procedure :: get_integral => mci_get_integral
    procedure :: get_error => mci_get_error
    procedure :: get_efficiency => mci_get_efficiency
    procedure :: get_time => mci_get_time
```

⟨*MCI base: procedures*⟩+≡
```
    function mci_get_integral (mci) result (integral)
      class(mci_t), intent(in) :: mci
      real(default) :: integral
      if (mci%integral_known) then
         integral = mci%integral
      else
         call msg_bug ("The integral is unknown. This is presumably a" // &
              "WHIZARD bug.")
      end if
    end function mci_get_integral

    function mci_get_error (mci) result (error)
```

```
      class(mci_t), intent(in) :: mci
      real(default) :: error
      if (mci%error_known) then
         error = mci%error
      else
         error = 0
      end if
   end function mci_get_error

   function mci_get_efficiency (mci) result (efficiency)
      class(mci_t), intent(in) :: mci
      real(default) :: efficiency
      if (mci%efficiency_known) then
         efficiency = mci%efficiency
      else
         efficiency = 0
      end if
   end function mci_get_efficiency

   function mci_get_time (mci) result (time)
      class(mci_t), intent(in) :: mci
      real(default) :: time
      if (mci%use_timer) then
         time = mci%timer
      else
         time = 0
      end if
   end function mci_get_time
```

Return the MD5 sum of the configuration. This may be overridden in an extension, to return a different MD5 sum.

⟨*MCI base: mci: TBP*⟩+≡
```
   procedure :: get_md5sum => mci_get_md5sum
```

⟨*MCI base: procedures*⟩+≡
```
   function mci_get_md5sum (mci) result (md5sum)
      class(mci_t), intent(in) :: mci
      character(32) :: md5sum
      md5sum = mci%md5sum
   end function mci_get_md5sum
```

13.1.2   MCI instance

The base type contains an array of channel weights. The value `mci_weight` is the combined MCI weight that corresponds to a particular sampling point.

For convenience, we also store the `x` and Jacobian values for this sampling point.

⟨*MCI base: public*⟩+≡
```
   public :: mci_instance_t
```

⟨*MCI base: types*⟩+≡
```
type, abstract :: mci_instance_t
  logical :: valid = .false.
  real(default), dimension(:), allocatable :: w
  real(default), dimension(:), allocatable :: f
  real(default), dimension(:,:), allocatable :: x
  integer :: selected_channel = 0
  real(default) :: mci_weight = 0
  real(default) :: integrand  = 0
 contains
 ⟨MCI base: mci instance: TBP⟩
 end type mci_instance_t
```

Output: deferred

⟨*MCI base: mci instance: TBP*⟩≡
```
procedure (mci_instance_write), deferred :: write
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
  subroutine mci_instance_write (object, unit, pacify)
    import
    class(mci_instance_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: pacify
  end subroutine mci_instance_write
end interface
```

A finalizer, just in case.

⟨*MCI base: mci instance: TBP*⟩+≡
```
procedure (mci_instance_final), deferred :: final
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
  subroutine mci_instance_final (object)
    import
    class(mci_instance_t), intent(inout) :: object
  end subroutine mci_instance_final
end interface
```

Init: basic initializer for the arrays, otherwise deferred. Assigning the `mci` object is also deferred, because it depends on the concrete type.

The weights are initialized with an uniform normalized value.

⟨*MCI base: mci instance: TBP*⟩+≡
```
procedure (mci_instance_base_init), deferred :: init
procedure :: base_init => mci_instance_base_init
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_instance_base_init (mci_instance, mci)
  class(mci_instance_t), intent(out) :: mci_instance
  class(mci_t), intent(in), target :: mci
  allocate (mci_instance%w (mci%n_channel))
  allocate (mci_instance%f (mci%n_channel))
  allocate (mci_instance%x (mci%n_dim, mci%n_channel))
```

```
      if (size (mci_instance%w) /= 0) then
         mci_instance%w = 1._default / size (mci_instance%w)
      end if
      mci_instance%f = 0
      mci_instance%x = 0
   end subroutine mci_instance_base_init
```

Compute the overall weight factor for a configuration of x values and Jacobians f. The x values come in `n_channel` rows with `n_dim` entries each. The f factors constitute an array with `n_channel` entries.

We assume that the x and f arrays are already stored inside the MC instance. The result is also stored there.

⟨*MCI base: mci instance: TBP*⟩+≡
```
   procedure (mci_instance_compute_weight), deferred :: compute_weight
```

⟨*MCI base: interfaces*⟩+≡
```
   abstract interface
      subroutine mci_instance_compute_weight (mci, c)
        import
        class(mci_instance_t), intent(inout) :: mci
        integer, intent(in) :: c
      end subroutine mci_instance_compute_weight
   end interface
```

Record the integrand as returned by the sampler. Depending on the implementation, this may merely copy the value, or do more complicated things.

We may need the MCI weight for the actual computations, so this should be called after the previous routine.

⟨*MCI base: mci instance: TBP*⟩+≡
```
   procedure (mci_instance_record_integrand), deferred :: record_integrand
```

⟨*MCI base: interfaces*⟩+≡
```
   abstract interface
      subroutine mci_instance_record_integrand (mci, integrand)
        import
        class(mci_instance_t), intent(inout) :: mci
        real(default), intent(in) :: integrand
      end subroutine mci_instance_record_integrand
   end interface
```

Sample a point directly: evaluate the sampler, then compute the weight and the weighted integrand. Finally, record the integrand within the MCI instance.

If a signal (interrupt) was raised recently, we abort the calculation before entering the sampler. Thus, a previous calculation will have completed and any data are already recorded, but any new point can be discarded. If the `abort` flag is present, we may delay the interrupt, so we can do some cleanup.

⟨*MCI base: mci instance: TBP*⟩+≡
```
   procedure :: evaluate => mci_instance_evaluate
```

⟨*MCI base: procedures*⟩+≡
```
   subroutine mci_instance_evaluate (mci, sampler, c, x)
     class(mci_instance_t), intent(inout) :: mci
```

```
      class(mci_sampler_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x
      real(default) :: val
      call sampler%evaluate (c, x, val, mci%x, mci%f)
      mci%valid = sampler%is_valid ()
      if (mci%valid) then
         call mci%compute_weight (c)
         call mci%record_integrand (val)
      end if
   end subroutine mci_instance_evaluate
```

Initiate and terminate simulation. In contrast to integration, we implement these as methods of the process instance, since the `mci` configuration object is unchanged.

The safety factor reduces the acceptance probability for unweighted events. The implementation of this feature depends on the concrete type.

⟨*MCI base: mci instance: TBP*⟩+≡
```
    procedure (mci_instance_init_simulation), deferred :: init_simulation
    procedure (mci_instance_final_simulation), deferred :: final_simulation
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
       subroutine mci_instance_init_simulation (instance, safety_factor)
         import
         class(mci_instance_t), intent(inout) :: instance
         real(default), intent(in), optional :: safety_factor
       end subroutine mci_instance_init_simulation
    end interface

    abstract interface
       subroutine mci_instance_final_simulation (instance)
         import
         class(mci_instance_t), intent(inout) :: instance
       end subroutine mci_instance_final_simulation
    end interface
```

Assuming that the sampler is in a completely defined state, just extract the data that `evaluate` would compute. Also record the integrand.

⟨*MCI base: mci instance: TBP*⟩+≡
```
    procedure :: fetch => mci_instance_fetch
```

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_instance_fetch (mci, sampler, c)
      class(mci_instance_t), intent(inout) :: mci
      class(mci_sampler_t), intent(in) :: sampler
      integer, intent(in) :: c
      real(default) :: val
      mci%valid = sampler%is_valid ()
      if (mci%valid) then
         call sampler%fetch (val, mci%x, mci%f)
         call mci%compute_weight (c)
         call mci%record_integrand (val)
```

```
      end if
   end subroutine mci_instance_fetch
```

The value, i.e., the weighted integrand, is the integrand (which should be taken as-is from the sampler) multiplied by the MCI weight.

⟨*MCI base: mci instance: TBP*⟩+≡
```
   procedure :: get_value => mci_instance_get_value
```

⟨*MCI base: procedures*⟩+≡
```
   function mci_instance_get_value (mci) result (value)
     class(mci_instance_t), intent(in) :: mci
     real(default) :: value
     if (mci%valid) then
        value = mci%integrand * mci%mci_weight
     else
        value = 0
     end if
   end function mci_instance_get_value
```

This is an extra routine. By default, the event weight is equal to the value returned by the previous routine. However, if we select a channel for event generation not just based on the channel weights, the event weight has to account for this bias, so the event weight that applies to event generation is different. In that case, we should override the default routine.

⟨*MCI base: mci instance: TBP*⟩+≡
```
   procedure :: get_event_weight => mci_instance_get_value
```

Excess weight can occur during unweighted event generation, if the assumed maximum value of the integrand is too small. This excess should be normalized in the same way as the event weight above (which for unweighted events becomes unity).

⟨*MCI base: mci instance: TBP*⟩+≡
```
   procedure (mci_instance_get_event_excess), deferred :: get_event_excess
```

⟨*MCI base: interfaces*⟩+≡
```
   abstract interface
     function mci_instance_get_event_excess (mci) result (excess)
       import
       class(mci_instance_t), intent(in) :: mci
       real(default) :: excess
     end function mci_instance_get_event_excess
   end interface
```

13.1.3 MCI state

This object can hold the relevant information that allows us to reconstruct the MCI instance without re-evaluating the sampler completely.

We store the `x_in` MC input parameter set, which coincides with the section of the complete `x` array that belongs to a particular channel. We also store the MC function value. When we want to reconstruct the state, we can use the input array to recover the complete `x` and `f` arrays (i.e., the kinematics), but do not need to recompute the MC function value (the dynamics).

The `mci_state_t` may be extended, to allow storing/recalling more information. In that case, we would override the type-bound procedures. However, the base type is also a concrete type and self-contained.

⟨*MCI base: public*⟩+≡
```
public :: mci_state_t
```

⟨*MCI base: types*⟩+≡
```
type :: mci_state_t
   integer :: selected_channel = 0
   real(default), dimension(:), allocatable :: x_in
   real(default) :: val
 contains
 ⟨MCI base: mci state: TBP⟩
 end type mci_state_t
```

Output:

⟨*MCI base: mci state: TBP*⟩≡
```
procedure :: write => mci_state_write
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_state_write (object, unit)
  class(mci_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)")  "MCI state:"
  write (u, "(3x,A,I0)")  "Channel   = ", object%selected_channel
  write (u, "(3x,A,999(1x,F12.10))")  "x (in)    =", object%x_in
  write (u, "(3x,A,ES19.12)")  "Integrand = ", object%val
end subroutine mci_state_write
```

To store the object, we take the relevant section of the `x` array. The channel used for storing data is taken from the `instance` object, but it could be arbitrary in principle.

⟨*MCI base: mci instance: TBP*⟩+≡
```
procedure :: store => mci_instance_store
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_instance_store (mci, state)
  class(mci_instance_t), intent(in) :: mci
  class(mci_state_t), intent(out) :: state
  state%selected_channel = mci%selected_channel
  allocate (state%x_in (size (mci%x, 1)))
  state%x_in = mci%x(:,mci%selected_channel)
  state%val = mci%integrand
end subroutine mci_instance_store
```

Recalling the state, we must consult the sampler in order to fully reconstruct the `x` and `f` arrays. The integrand value is known, and we also give it to the sampler, bypassing evaluation.

The final steps are equivalent to the `evaluate` method above.

⟨*MCI base: mci instance: TBP*⟩+≡
```
procedure :: recall => mci_instance_recall
```

1596

⟨*MCI base: procedures*⟩+≡
```
    subroutine mci_instance_recall (mci, sampler, state)
      class(mci_instance_t), intent(inout) :: mci
      class(mci_sampler_t), intent(inout) :: sampler
      class(mci_state_t), intent(in) :: state
      if (size (state%x_in) == size (mci%x, 1) &
           .and. state%selected_channel <= size (mci%x, 2)) then
         call sampler%rebuild (state%selected_channel, &
              state%x_in, state%val, mci%x, mci%f)
         call mci%compute_weight (state%selected_channel)
         call mci%record_integrand (state%val)
      else
         call msg_fatal ("Recalling event: mismatch in channel or dimension")
      end if
    end subroutine mci_instance_recall
```

13.1.4 MCI sampler

A sampler is an object that implements a multi-channel parameterization of the unit hypercube. Specifically, it is able to compute, given a channel and a set of x MC parameter values, a the complete set of x values and associated Jacobian factors f for all channels.

Furthermore, the sampler should return a single real value, the integrand, for the given point in the hypercube.

It must implement a method `evaluate` for performing the above computations.

⟨*MCI base: public*⟩+≡
```
    public :: mci_sampler_t
```

⟨*MCI base: types*⟩+≡
```
    type, abstract :: mci_sampler_t
     contains
     ⟨MCI base: mci sampler: TBP⟩
    end type mci_sampler_t
```

Output, deferred to the implementation.

⟨*MCI base: mci sampler: TBP*⟩≡
```
    procedure (mci_sampler_write), deferred :: write
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
      subroutine mci_sampler_write (object, unit, testflag)
        import
        class(mci_sampler_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: testflag
      end subroutine mci_sampler_write
    end interface
```

The evaluation routine. Input is the channel index `c` and the one-dimensional parameter array `x_in`. Output are the integrand value `val`, the two-dimensional parameter array `x` and the Jacobian array `f`.

⟨*MCI base: mci sampler: TBP*⟩+≡
```
procedure (mci_sampler_evaluate), deferred :: evaluate
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_sampler_evaluate (sampler, c, x_in, val, x, f)
     import
     class(mci_sampler_t), intent(inout) :: sampler
     integer, intent(in) :: c
     real(default), dimension(:), intent(in) :: x_in
     real(default), intent(out) :: val
     real(default), dimension(:,:), intent(out) :: x
     real(default), dimension(:), intent(out) :: f
   end subroutine mci_sampler_evaluate
end interface
```

Query the validity of the sampling point. Can be called after `evaluate`.

⟨*MCI base: mci sampler: TBP*⟩+≡
```
procedure (mci_sampler_is_valid), deferred :: is_valid
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   function mci_sampler_is_valid (sampler) result (valid)
     import
     class(mci_sampler_t), intent(in) :: sampler
     logical :: valid
   end function mci_sampler_is_valid
end interface
```

The shortcut. Again, the channel index `c` and the parameter array `x_in` are input. However, we also provide the integrand value `val`, and we just require that the complete parameter array `x` and Jacobian array `f` are recovered.

⟨*MCI base: mci sampler: TBP*⟩+≡
```
procedure (mci_sampler_rebuild), deferred :: rebuild
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_sampler_rebuild (sampler, c, x_in, val, x, f)
     import
     class(mci_sampler_t), intent(inout) :: sampler
     integer, intent(in) :: c
     real(default), dimension(:), intent(in) :: x_in
     real(default), intent(in) :: val
     real(default), dimension(:,:), intent(out) :: x
     real(default), dimension(:), intent(out) :: f
   end subroutine mci_sampler_rebuild
end interface
```

This routine should extract the important data from a sampler that has been filled by other means. We fetch the integrand value `val`, the two-dimensional parameter array `x` and the Jacobian array `f`.

⟨*MCI base: mci sampler: TBP*⟩+≡
```
    procedure (mci_sampler_fetch), deferred :: fetch
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
      subroutine mci_sampler_fetch (sampler, val, x, f)
        import
        class(mci_sampler_t), intent(in) :: sampler
        real(default), intent(out) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
      end subroutine mci_sampler_fetch
    end interface
```

13.1.5 Results record

This is an abstract type which allows us to implement callback: each integration results can optionally be recorded to an instance of this object. The actual object may store a new result, average results, etc. It may also display a result on-line or otherwise, whenever the `record` method is called.

⟨*MCI base: public*⟩+≡
```
    public :: mci_results_t
```

⟨*MCI base: types*⟩+≡
```
    type, abstract :: mci_results_t
     contains
     ⟨MCI base: mci results: TBP⟩
    end type mci_results_t
```

The output routine is deferred. We provide an extra `verbose` flag, which could serve any purpose.

⟨*MCI base: mci results: TBP*⟩≡
```
    procedure (mci_results_write), deferred :: write
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
      subroutine mci_results_write (object, unit, verbose, suppress)
        import
        class(mci_results_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: verbose, suppress
      end subroutine mci_results_write
    end interface
```

This is the `record` method, which can be called directly from the integrator.

⟨*MCI base: mci results: TBP*⟩+≡
```
    procedure (mci_results_record), deferred :: record
```

⟨*MCI base: interfaces*⟩+≡
```
    abstract interface
        subroutine mci_results_record (object, n_it, &
             n_calls, integral, error, efficiency, chain_weights, suppress)
          import
          class(mci_results_t), intent(inout) :: object
          integer, intent(in) :: n_it
          integer, intent(in) :: n_calls
          real(default), intent(in) :: integral
          real(default), intent(in) :: error
          real(default), intent(in) :: efficiency
          real(default), dimension(:), intent(in), optional :: chain_weights
          logical, intent(in), optional :: suppress
        end subroutine mci_results_record
    end interface
```

13.1.6   Unit tests

⟨*MCI base: public*⟩+≡
```
    public :: mci_base_test
```

⟨*MCI base: tests*⟩≡
```
    subroutine mci_base_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*MCI base: execute tests*⟩
```
    end subroutine mci_base_test
```

Test implementation of the configuration type

The concrete type contains the number of requested calls and the integral result, to be determined.

The `max_factor` entry is set for the actual test integration, where the integrand is not unity but some other constant value. This value should be set here, such that the actual maximum of the integrand is known when vetoing unweighted events.

⟨*MCI base: public*⟩+≡
```
    public :: mci_test_t
```

⟨*MCI base: test types*⟩≡
```
    type, extends (mci_t) :: mci_test_t
       integer :: divisions = 0
       integer :: tries = 0
       real(default) :: max_factor = 1
     contains
       procedure :: final => mci_test_final
       procedure :: write => mci_test_write
       procedure :: startup_message => mci_test_startup_message
       procedure :: declare_flat_dimensions => mci_test_ignore_flat_dimensions
       procedure :: declare_equivalences => mci_test_ignore_equivalences
       procedure :: set_divisions => mci_test_set_divisions
       procedure :: set_max_factor => mci_test_set_max_factor
```

```
      procedure :: allocate_instance => mci_test_allocate_instance
      procedure :: integrate => mci_test_integrate
      procedure :: prepare_simulation => mci_test_ignore_prepare_simulation
      procedure :: generate_weighted_event => mci_test_generate_weighted_event
      procedure :: generate_unweighted_event => &
           mci_test_generate_unweighted_event
      procedure :: rebuild_event => mci_test_rebuild_event
    end type mci_test_t
```

Finalizer: base version is sufficient

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_final (object)
      class(mci_test_t), intent(inout) :: object
      call object%base_final ()
    end subroutine mci_test_final
```

Output: trivial

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_write (object, unit, pacify, md5sum_version)
      class(mci_test_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      logical, intent(in), optional :: md5sum_version
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "Test integrator:"
      call object%base_write (u, pacify, md5sum_version)
      if (object%divisions /= 0) then
         write (u, "(3x,A,I0)")  "Number of divisions  = ", object%divisions
      end if
      if (allocated (object%rng))  call object%rng%write (u)
    end subroutine mci_test_write
```

Short version.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_startup_message (mci, unit, n_calls)
      class(mci_test_t), intent(in) :: mci
      integer, intent(in), optional :: unit, n_calls
      call mci%base_startup_message (unit = unit, n_calls = n_calls)
      write (msg_buffer, "(A,1x,I0,1x,A)") &
           "Integrator: Test:", mci%divisions, "divisions"
      call msg_message (unit = unit)
    end subroutine mci_test_startup_message
```

This is a no-op for the test integrator.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_ignore_flat_dimensions (mci, dim_flat)
      class(mci_test_t), intent(inout) :: mci
      integer, dimension(:), intent(in) :: dim_flat
    end subroutine mci_test_ignore_flat_dimensions
```

Ditto.

⟨MCI base: tests⟩+≡

```
    subroutine mci_test_ignore_equivalences (mci, channel, dim_offset)
      class(mci_test_t), intent(inout) :: mci
      type(phs_channel_t), dimension(:), intent(in) :: channel
      integer, intent(in) :: dim_offset
    end subroutine mci_test_ignore_equivalences
```

Set the number of divisions to a nonzero value.

⟨MCI base: tests⟩+≡

```
    subroutine mci_test_set_divisions (object, divisions)
      class(mci_test_t), intent(inout) :: object
      integer, intent(in) :: divisions
      object%divisions = divisions
    end subroutine mci_test_set_divisions
```

Set the maximum factor (default is 1).

⟨MCI base: tests⟩+≡

```
    subroutine mci_test_set_max_factor (object, max_factor)
      class(mci_test_t), intent(inout) :: object
      real(default), intent(in) :: max_factor
      object%max_factor = max_factor
    end subroutine mci_test_set_max_factor
```

Allocate instance with matching type.

⟨MCI base: tests⟩+≡

```
    subroutine mci_test_allocate_instance (mci, mci_instance)
      class(mci_test_t), intent(in) :: mci
      class(mci_instance_t), intent(out), pointer :: mci_instance
      allocate (mci_test_instance_t :: mci_instance)
    end subroutine mci_test_allocate_instance
```

Integrate: sample at the midpoints of uniform bits and add the results. We implement this for one and for two dimensions. In the latter case, we scan over two channels and multiply with the channel weights.

The arguments `n_it` and `n_calls` are ignored in this implementations.

The test integrator does not set error or efficiency, so those will remain undefined.

⟨MCI base: tests⟩+≡

```
    subroutine mci_test_integrate (mci, instance, sampler, &
         n_it, n_calls, results, pacify)
      class(mci_test_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      integer, intent(in) :: n_it
      integer, intent(in) :: n_calls
      logical, intent(in), optional :: pacify
      class(mci_results_t), intent(inout), optional :: results
      real(default), dimension(:), allocatable :: integral
      real(default), dimension(:), allocatable :: x
      integer :: i, j, c
```

1602

```
      select type (instance)
      type is (mci_test_instance_t)
         allocate (integral (mci%n_channel))
         integral = 0
         allocate (x (mci%n_dim))
         select case (mci%n_dim)
         case (1)
            do c = 1, mci%n_channel
               do i = 1, mci%divisions
                  x(1) = (i - 0.5_default) / mci%divisions
                  call instance%evaluate (sampler, c, x)
                  integral(c) = integral(c) + instance%get_value ()
               end do
            end do
            mci%integral = dot_product (instance%w, integral) &
                 / mci%divisions
            mci%integral_known = .true.
         case (2)
            do c = 1, mci%n_channel
               do i = 1, mci%divisions
                  x(1) = (i - 0.5_default) / mci%divisions
                  do j = 1, mci%divisions
                     x(2) = (j - 0.5_default) / mci%divisions
                     call instance%evaluate (sampler, c, x)
                     integral(c) = integral(c) + instance%get_value ()
                  end do
               end do
            end do
            mci%integral = dot_product (instance%w, integral) &
                 / mci%divisions / mci%divisions
            mci%integral_known = .true.
         end select
         if (present (results)) then
            call results%record (n_it, n_calls, &
                 mci%integral, mci%error, &
                 efficiency = 0._default)
         end if
      end select
    end subroutine mci_test_integrate
```

Simulation initializer and finalizer: nothing to do here.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_ignore_prepare_simulation (mci)
      class(mci_test_t), intent(inout) :: mci
    end subroutine mci_test_ignore_prepare_simulation
```

Event generator. We use mock random numbers for first selecting the channel and then setting the x values. The results reside in the state of `instance` and `sampler`.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_generate_weighted_event (mci, instance, sampler)
      class(mci_test_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout), target :: instance
```

1603

```
            class(mci_sampler_t), intent(inout), target :: sampler
            real(default) :: r
            real(default), dimension(:), allocatable :: x
            integer :: c
            select type (instance)
            type is (mci_test_instance_t)
               allocate (x (mci%n_dim))
               select case (mci%n_channel)
               case (1)
                  c = 1
                  call mci%rng%generate (x(1))
               case (2)
                  call mci%rng%generate (r)
                  if (r < instance%w(1)) then
                     c = 1
                  else
                     c = 2
                  end if
                  call mci%rng%generate (x)
               end select
               call instance%evaluate (sampler, c, x)
            end select
         end subroutine mci_test_generate_weighted_event
```

For unweighted events, we generate weighted events and apply a simple rejection step to the relative event weight, until an event passes.

(This might result in an endless loop if we happen to be in sync with the mock random generator cycle. Therefore, limit the number of tries.)

⟨*MCI base: tests*⟩+≡
```
         subroutine mci_test_generate_unweighted_event (mci, instance, sampler)
            class(mci_test_t), intent(inout) :: mci
            class(mci_instance_t), intent(inout), target :: instance
            class(mci_sampler_t), intent(inout), target :: sampler
            real(default) :: r
            integer :: i
            select type (instance)
            type is (mci_test_instance_t)
               mci%tries = 0
               do i = 1, 10
                  call mci%generate_weighted_event (instance, sampler)
                  mci%tries = mci%tries + 1
                  call mci%rng%generate (r)
                  if (r < instance%rel_value)  exit
               end do
            end select
         end subroutine mci_test_generate_unweighted_event
```

Here, we rebuild the event from the state without consulting the rng.

⟨*MCI base: tests*⟩+≡
```
         subroutine mci_test_rebuild_event (mci, instance, sampler, state)
            class(mci_test_t), intent(inout) :: mci
            class(mci_instance_t), intent(inout) :: instance
            class(mci_sampler_t), intent(inout) :: sampler
```

```
      class(mci_state_t), intent(in) :: state
      select type (instance)
      type is (mci_test_instance_t)
         call instance%recall (sampler, state)
      end select
    end subroutine mci_test_rebuild_event
```

Instance of the test MCI type

This instance type simulates the VAMP approach. We implement the VAMP multi-channel formula, but keep the channel-specific probability functions g_i smooth and fixed. We also keep the weights fixed.

The setup is as follows: we have n mappings of the unit hypercube

$$x = x(x^{(k)}) \qquad \text{where } x = (x_1, \ldots). \tag{13.1}$$

The Jacobian factors are the determinants

$$f^{(k)}(x^{(k)}) = \left| \frac{\partial x}{\partial x^{(k)}} \right| \tag{13.2}$$

We introduce arbitrary probability functions

$$g^{(k)}(x^{(k)}) \qquad \text{with} \quad \int dx^{(k)} g^{(k)}(x^{(k)}) = 1 \tag{13.3}$$

and weights

$$w_k \qquad \text{with} \quad \sum_k w_k = 1 \tag{13.4}$$

and construct the joint probability function

$$g(x) = \sum_k w_k \frac{g^{(k)}(x^{(k)}(x))}{f^{(k)}(x^{(k)}(x))} \tag{13.5}$$

which also satisfies

$$\int g(x)\, dx = 1. \tag{13.6}$$

The algorithm implements a resolution of unity as follows

$$
\begin{aligned}
1 = \int dx &= \int \frac{g(x)}{g(x)} dx \\
&= \sum w_k \int \frac{g^{(k)}(x^{(k)}(x))}{f^{(k)}(x^{(k)}(x))} \frac{dx}{g(x)} \\
&= \sum w_k \int g^{(k)}(x^{(k)}) \frac{dx^{(k)}}{g(x(x^{(k)}))}
\end{aligned}
\tag{13.7}
$$

where each of the integrals in the sum is evaluated using the channel-specific variables $x^{(k)}$.

We provide two examples: (1) trivial with one channel, one dimension, and all functions unity and (2) two channels and two dimensions with

$$x(x^{(1)}) = (x_1^{(1)}, x_2^{(1)})$$
$$x(x^{(2)}) = (x_1^{(2)2}, x_2^{(2)}) \tag{13.8}$$

hence

$$f^{(1)} \equiv 1, \qquad\qquad f^{(2)}(x^{(2)}) = 2x_1^{(2)} \tag{13.9}$$

The probability functions are

$$g^{(1)} \equiv 1, \qquad\qquad g^{(2)}(x^{(2)}) = 2x_2^{(2)} \tag{13.10}$$

In the concrete implementation of the integrator instance we store values for the channel probabilities g_i and the accumulated probability g.

We also store the result (product of integrand and MCI weight), the expected maximum for the result in each channel.

⟨*MCI base: public*⟩+≡
```
    public :: mci_test_instance_t
```

⟨*MCI base: test types*⟩+≡
```
    type, extends (mci_instance_t) :: mci_test_instance_t
      type(mci_test_t), pointer :: mci => null ()
      real(default) :: g = 0
      real(default), dimension(:), allocatable :: gi
      real(default) :: value = 0
      real(default) :: rel_value = 0
      real(default), dimension(:), allocatable :: max
    contains
      procedure :: write => mci_test_instance_write
      procedure :: final => mci_test_instance_final
      procedure :: init => mci_test_instance_init
      procedure :: compute_weight => mci_test_instance_compute_weight
      procedure :: record_integrand => mci_test_instance_record_integrand
      procedure :: init_simulation => mci_test_instance_init_simulation
      procedure :: final_simulation => mci_test_instance_final_simulation
      procedure :: get_event_excess => mci_test_instance_get_event_excess
    end type mci_test_instance_t
```

Output: trivial

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_instance_write (object, unit, pacify)
      class(mci_test_instance_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      integer :: u, c
      u = output_unit (unit)
      write (u, "(1x,A,ES13.7)") "Result value = ", object%value
      write (u, "(1x,A,ES13.7)") "Rel. weight  = ", object%rel_value
      write (u, "(1x,A,ES13.7)") "Integrand    = ", object%integrand
      write (u, "(1x,A,ES13.7)") "MCI weight   = ", object%mci_weight
      write (u, "(3x,A,I0)")  "c = ", object%selected_channel
```

```
      write (u, "(3x,A,ES13.7)") "g = ", object%g
      write (u, "(1x,A)")  "Channel parameters:"
      do c = 1, object%mci%n_channel
         write (u, "(1x,I0,A,4(1x,ES13.7))")  c, ": w/f/g/m =", &
              object%w(c), object%f(c), object%gi(c), object%max(c)
         write (u, "(4x,A,9(1x,F9.7))")  "x =", object%x(:,c)
      end do
    end subroutine mci_test_instance_write
```

The finalizer is empty.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_instance_final (object)
      class(mci_test_instance_t), intent(inout) :: object
    end subroutine mci_test_instance_final
```

Initializer. We make use of the analytical result that the maximum of the weighted integrand, in each channel, is equal to 1 (one-dimensional case) and 2 (two-dimensional case), respectively.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_instance_init (mci_instance, mci)
      class(mci_test_instance_t), intent(out) :: mci_instance
      class(mci_t), intent(in), target :: mci
      call mci_instance%base_init (mci)
      select type (mci)
      type is (mci_test_t)
         mci_instance%mci => mci
      end select
      allocate (mci_instance%gi (mci%n_channel))
      mci_instance%gi = 0
      allocate (mci_instance%max (mci%n_channel))
      select case (mci%n_channel)
      case (1)
         mci_instance%max = 1._default
      case (2)
         mci_instance%max = 2._default
      end select
    end subroutine mci_test_instance_init
```

Compute weight: we implement the VAMP multi-channel formula. The channel probabilities `gi` are predefined functions.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_instance_compute_weight (mci, c)
      class(mci_test_instance_t), intent(inout) :: mci
      integer, intent(in) :: c
      integer :: i
      mci%selected_channel = c
      select case (mci%mci%n_dim)
      case (1)
         mci%gi(1) = 1
      case (2)
         mci%gi(1) = 1
         mci%gi(2) = 2 * mci%x(2,2)
```

```
      end select
      mci%g = 0
      do i = 1, mci%mci%n_channel
          mci%g = mci%g + mci%w(i) * mci%gi(i) / mci%f(i)
      end do
      mci%mci_weight = mci%gi(c) / mci%g
    end subroutine mci_test_instance_compute_weight
```

Record the integrand. Apply the Jacobian weight to get the absolute value. Divide by the channel maximum and by any overall factor to get the value relative to the maximum.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_instance_record_integrand (mci, integrand)
      class(mci_test_instance_t), intent(inout) :: mci
      real(default), intent(in) :: integrand
      mci%integrand = integrand
      mci%value = mci%integrand * mci%mci_weight
      mci%rel_value = mci%value / mci%max(mci%selected_channel) &
            / mci%mci%max_factor
    end subroutine mci_test_instance_record_integrand
```

Nothing to do here.

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_instance_init_simulation (instance, safety_factor)
      class(mci_test_instance_t), intent(inout) :: instance
      real(default), intent(in), optional :: safety_factor
    end subroutine mci_test_instance_init_simulation

    subroutine mci_test_instance_final_simulation (instance)
      class(mci_test_instance_t), intent(inout) :: instance
    end subroutine mci_test_instance_final_simulation
```

Return always zero.

⟨*MCI base: tests*⟩+≡
```
    function mci_test_instance_get_event_excess (mci) result (excess)
      class(mci_test_instance_t), intent(in) :: mci
      real(default) :: excess
      excess = 0
    end function mci_test_instance_get_event_excess
```

Test sampler

The test sampler implements a fixed configuration, either trivial (one-channel, one-dimension), or slightly nontrivial (two-channel, two-dimension). In the second channel, the first parameter is mapped according to $x_1 = x_1^{(2)2}$, so we have $f^{(2)}(x^{(2)}) = 2x_1^{(2)}$.

For display purposes, we store the return values inside the object. This is not strictly necessary.

⟨*MCI base: test types*⟩+≡
```
    type, extends (mci_sampler_t) :: test_sampler_t
```

```
     real(default) :: integrand = 0
     integer :: selected_channel = 0
     real(default), dimension(:,:), allocatable :: x
     real(default), dimension(:), allocatable :: f
   contains
     procedure :: init => test_sampler_init
     procedure :: write => test_sampler_write
     procedure :: compute => test_sampler_compute
     procedure :: is_valid => test_sampler_is_valid
     procedure :: evaluate => test_sampler_evaluate
     procedure :: rebuild => test_sampler_rebuild
     procedure :: fetch => test_sampler_fetch
   end type test_sampler_t
```

⟨*MCI base: tests*⟩+≡
```
   subroutine test_sampler_init (sampler, n)
     class(test_sampler_t), intent(out) :: sampler
     integer, intent(in) :: n
     allocate (sampler%x (n, n))
     allocate (sampler%f (n))
   end subroutine test_sampler_init
```

Output
⟨*MCI base: tests*⟩+≡
```
   subroutine test_sampler_write (object, unit, testflag)
     class(test_sampler_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: testflag
     integer :: u, c
     u = output_unit (unit)
     write (u, "(1x,A)")  "Test sampler:"
     write (u, "(3x,A,ES13.7)")  "Integrand = ", object%integrand
     write (u, "(3x,A,I0)")      "Channel   = ", object%selected_channel
     do c = 1, size (object%f)
        write (u, "(1x,I0,':',1x,A,ES13.7)") c, "f = ", object%f(c)
        write (u, "(4x,A,9(1x,F9.7))") "x =", object%x(:,c)
     end do
   end subroutine test_sampler_write
```

Compute x and Jacobians, given the input parameter array. This is called both
by `evaluate` and `rebuild`.
⟨*MCI base: tests*⟩+≡
```
   subroutine test_sampler_compute (sampler, c, x_in)
     class(test_sampler_t), intent(inout) :: sampler
     integer, intent(in) :: c
     real(default), dimension(:), intent(in) :: x_in
     sampler%selected_channel = c
     select case (size (sampler%f))
     case (1)
        sampler%x(:,1) = x_in
        sampler%f = 1
     case (2)
```

```
        select case (c)
        case (1)
           sampler%x(:,1) = x_in
           sampler%x(1,2) = sqrt (x_in(1))
           sampler%x(2,2) = x_in(2)
        case (2)
           sampler%x(1,1) = x_in(1) ** 2
           sampler%x(2,1) = x_in(2)
           sampler%x(:,2) = x_in
        end select
        sampler%f(1) = 1
        sampler%f(2) = 2 * sampler%x(1,2)
     end select
  end subroutine test_sampler_compute
```

The point is always valid.

⟨*MCI base: tests*⟩+≡
```
  function test_sampler_is_valid (sampler) result (valid)
    class(test_sampler_t), intent(in) :: sampler
    logical :: valid
    valid = .true.
  end function test_sampler_is_valid
```

The integrand is always equal to 1.

⟨*MCI base: tests*⟩+≡
```
  subroutine test_sampler_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    sampler%integrand = 1
    val = sampler%integrand
    x = sampler%x
    f = sampler%f
  end subroutine test_sampler_evaluate
```

Construct kinematics from the input x array. Set the integrand instead of evaluating it.

⟨*MCI base: tests*⟩+≡
```
  subroutine test_sampler_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    sampler%integrand = val
    x = sampler%x
```

```
    f = sampler%f
  end subroutine test_sampler_rebuild
```

Recall contents.

⟨*MCI base: tests*⟩+≡
```
  subroutine test_sampler_fetch (sampler, val, x, f)
    class(test_sampler_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%integrand
    x = sampler%x
    f = sampler%f
  end subroutine test_sampler_fetch
```

Test results object

This mock object just stores and displays the current result.

⟨*MCI base: test types*⟩+≡
```
  type, extends (mci_results_t) :: mci_test_results_t
     integer :: n_it = 0
     integer :: n_calls = 0
     real(default) :: integral = 0
     real(default) :: error = 0
     real(default) :: efficiency = 0
   contains
   ⟨MCI base: mci test results: TBP⟩
  end type mci_test_results_t
```

Output.

⟨*MCI base: mci test results: TBP*⟩≡
```
  procedure :: write => mci_test_results_write
```

⟨*MCI base: tests*⟩+≡
```
  subroutine mci_test_results_write (object, unit, verbose, suppress)
    class(mci_test_results_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose, suppress
    integer :: u
    u = output_unit (unit)
    write (u, "(3x,A,1x,I0)") "Iterations = ", object%n_it
    write (u, "(3x,A,1x,I0)") "Calls      = ", object%n_calls
    write (u, "(3x,A,1x,F12.10)")  "Integral   = ", object%integral
    write (u, "(3x,A,1x,F12.10)")  "Error      = ", object%error
    write (u, "(3x,A,1x,F12.10)")  "Efficiency = ", object%efficiency
  end subroutine mci_test_results_write
```

Record result.

⟨*MCI base: mci test results: TBP*⟩+≡
```
  procedure :: record => mci_test_results_record
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_test_results_record (object, n_it, n_calls, &
        integral, error, efficiency, chain_weights, suppress)
      class(mci_test_results_t), intent(inout) :: object
      integer, intent(in) :: n_it
      integer, intent(in) :: n_calls
      real(default), intent(in) :: integral
      real(default), intent(in) :: error
      real(default), intent(in) :: efficiency
      real(default), dimension(:), intent(in), optional :: chain_weights
      logical, intent(in), optional :: suppress
      object%n_it = n_it
      object%n_calls = n_calls
      object%integral = integral
      object%error = error
      object%efficiency = efficiency
    end subroutine mci_test_results_record
```

Integrator configuration data

Construct and display a test integrator configuration object.

⟨*MCI base: execute tests*⟩≡
```
    call test (mci_base_1, "mci_base_1", &
        "integrator configuration", &
        u, results)
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_base_1 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler

      real(default) :: integrand

      write (u, "(A)")  "* Test output: mci_base_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
          &test integrator"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_test_t :: mci)
      call mci%set_dimensions (2, 2)

      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize instance"
      write (u, "(A)")

      call mci%allocate_instance (mci_instance)
```

1612

```
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_t :: sampler)
select type (sampler)
type is (test_sampler_t)
   call sampler%init (2)
end select

write (u, "(A)")  "* Evaluate sampler for given point and channel"
write (u, "(A)")

call sampler%evaluate (1, [0.25_default, 0.8_default], &
     integrand, mci_instance%x, mci_instance%f)

call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute MCI weight"
write (u, "(A)")

call mci_instance%compute_weight (1)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Get integrand and compute weight for another point"
write (u, "(A)")

call mci_instance%evaluate (sampler, 2, [0.5_default, 0.6_default])
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recall results, again"
write (u, "(A)")

call mci_instance%final ()
deallocate (mci_instance)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

call mci_instance%fetch (sampler, 2)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Retrieve value"
write (u, "(A)")

write (u, "(1x,A,ES13.7)")  "Weighted integrand = ", &
     mci_instance%get_value ()

call mci_instance%final ()
```

```
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_base_1"

   end subroutine mci_base_1
```

Trivial integral

Use the MCI approach to compute a trivial one-dimensional integral.

⟨*MCI base: execute tests*⟩+≡
```
      call test (mci_base_2, "mci_base_2", &
           "integration", &
           u, results)
```
⟨*MCI base: tests*⟩+≡
```
   subroutine mci_base_2 (u)
     integer, intent(in) :: u
     class(mci_t), allocatable, target :: mci
     class(mci_instance_t), pointer :: mci_instance => null ()
     class(mci_sampler_t), allocatable :: sampler

     write (u, "(A)")  "* Test output: mci_base_2"
     write (u, "(A)")  "*   Purpose: perform a test integral"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize integrator"
     write (u, "(A)")

     allocate (mci_test_t :: mci)
     call mci%set_dimensions (1, 1)
     select type (mci)
     type is (mci_test_t)
        call mci%set_divisions (10)
     end select

     call mci%write (u)

     write (u, "(A)")
     write (u, "(A)")  "* Initialize instance"
     write (u, "(A)")

     call mci%allocate_instance (mci_instance)
     call mci_instance%init (mci)

     write (u, "(A)")  "* Initialize test sampler"
     write (u, "(A)")

     allocate (test_sampler_t :: sampler)
     select type (sampler)
     type is (test_sampler_t)
        call sampler%init (1)
     end select
```

```
        write (u, "(A)")  "* Integrate"
        write (u, "(A)")

        call mci%integrate (mci_instance, sampler, 0, 0)

        call mci%write (u)

        call mci_instance%final ()
        call mci%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: mci_base_2"

    end subroutine mci_base_2
```

Nontrivial integral

Use the MCI approach to compute a simple two-dimensional integral with two channels.

⟨*MCI base: execute tests*⟩+≡
```
    call test (mci_base_3, "mci_base_3", &
         "integration (two channels)", &
         u, results)
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_base_3 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler

      write (u, "(A)")  "* Test output: mci_base_3"
      write (u, "(A)")  "*   Purpose: perform a nontrivial test integral"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_test_t :: mci)
      call mci%set_dimensions (2, 2)
      select type (mci)
      type is (mci_test_t)
         call mci%set_divisions (10)
      end select

      write (u, "(A)")  "* Initialize instance"
      write (u, "(A)")

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      write (u, "(A)")  "* Initialize test sampler"
```

1615

```
      write (u, "(A)")

      allocate (test_sampler_t :: sampler)
      select type (sampler)
      type is (test_sampler_t)
         call sampler%init (2)
      end select

      write (u, "(A)")  "* Integrate"
      write (u, "(A)")

      call mci%integrate (mci_instance, sampler, 0, 0)
      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Integrate with higher resolution"
      write (u, "(A)")

      select type (mci)
      type is (mci_test_t)
         call mci%set_divisions (100)
      end select

      call mci%integrate (mci_instance, sampler, 0, 0)
      call mci%write (u)

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_base_3"

    end subroutine mci_base_3
```

Event generation

We generate "random" events, one weighted and one unweighted. The test
implementation does not require an integration pass, we can generate events
immediately.

⟨*MCI base: execute tests*⟩+≡
```
    call test (mci_base_4, "mci_base_4", &
        "event generation (two channels)", &
        u, results)
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_base_4 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_base_4"
```

1616

```
write (u, "(A)")  "*   Purpose: generate events"
write (u, "(A)")

write (u, "(A)")  "* Initialize integrator, instance, sampler"
write (u, "(A)")

allocate (mci_test_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_test_t)
   call mci%set_divisions (10)
end select

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_t :: sampler)
select type (sampler)
type is (test_sampler_t)
   call sampler%init (2)
end select

allocate (rng_test_t :: rng)
call mci%import_rng (rng)

write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

call mci%generate_weighted_event (mci_instance, sampler)

call sampler%write (u)
write (u, *)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate unweighted event"
write (u, "(A)")

call mci%generate_unweighted_event (mci_instance, sampler)

select type (mci)
type is (mci_test_t)
   write (u, "(A,I0)")  " Success in try ", mci%tries
   write (u, "(A)")
end select

call sampler%write (u)
write (u, *)
call mci_instance%write (u)

call mci_instance%final ()
call mci%final ()

write (u, "(A)")
```

```
        write (u, "(A)")  "* Test output end: mci_base_4"

    end subroutine mci_base_4
```

Store and recall data

We generate an event and store the relevant data, i.e., the input parameters and
the result value for a particular channel. Then we use those data to recover the
event, as far as the MCI record is concerned.

⟨*MCI base: execute tests*⟩+≡
```
    call test (mci_base_5, "mci_base_5", &
         "store and recall", &
         u, results)
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_base_5 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng
      class(mci_state_t), allocatable :: state

      write (u, "(A)")  "* Test output: mci_base_5"
      write (u, "(A)")  "*   Purpose: store and recall an event"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator, instance, sampler"
      write (u, "(A)")

      allocate (mci_test_t :: mci)
      call mci%set_dimensions (2, 2)
      select type (mci)
      type is (mci_test_t)
         call mci%set_divisions (10)
      end select

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      allocate (test_sampler_t :: sampler)
      select type (sampler)
      type is (test_sampler_t)
         call sampler%init (2)
      end select

      allocate (rng_test_t :: rng)
      call mci%import_rng (rng)

      write (u, "(A)")  "* Generate weighted event"
      write (u, "(A)")

      call mci%generate_weighted_event (mci_instance, sampler)
```

```
        call sampler%write (u)
        write (u, *)
        call mci_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Store data"
        write (u, "(A)")

        allocate (state)
        call mci_instance%store (state)
        call mci_instance%final ()
        deallocate (mci_instance)

        call state%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Recall data and rebuild event"
        write (u, "(A)")

        call mci%allocate_instance (mci_instance)
        call mci_instance%init (mci)
        call mci%rebuild_event (mci_instance, sampler, state)

        call sampler%write (u)
        write (u, *)
        call mci_instance%write (u)

        call mci_instance%final ()
        call mci%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: mci_base_5"

    end subroutine mci_base_5
```

Chained channels

Chain channels together. In the base configuration, this just fills entries in an extra array (each channel may belong to a chain). In type implementations, this will be used for grouping equivalent channels by keeping their weights equal.

⟨*MCI base: execute tests*⟩+≡
```
    call test (mci_base_6, "mci_base_6", &
        "chained channels", &
        u, results)
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_base_6 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci

      write (u, "(A)")  "* Test output: mci_base_6"
      write (u, "(A)")  "*   Purpose: initialize and display &
```

```
      &test integrator with chains"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (1, 5)

    write (u, "(A)")  "* Introduce chains"
    write (u, "(A)")

    call mci%declare_chains ([1, 2, 2, 1, 2])

    call mci%write (u)

    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_6"

  end subroutine mci_base_6
```

Recording results

Compute a simple two-dimensional integral and record the result.

⟨*MCI base: execute tests*⟩+≡
```
    call test (mci_base_7, "mci_base_7", &
         "recording results", &
         u, results)
```
⟨*MCI base: tests*⟩+≡
```
  subroutine mci_base_7 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(mci_results_t), allocatable :: results

    write (u, "(A)")  "* Test output: mci_base_7"
    write (u, "(A)")  "*   Purpose: perform a nontrivial test integral &
         &and record results"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_test_t)
       call mci%set_divisions (10)
    end select
```

```
      write (u, "(A)")  "* Initialize instance"
      write (u, "(A)")

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      write (u, "(A)")  "* Initialize test sampler"
      write (u, "(A)")

      allocate (test_sampler_t :: sampler)
      select type (sampler)
      type is (test_sampler_t)
         call sampler%init (2)
      end select

      allocate (mci_test_results_t :: results)

      write (u, "(A)")  "* Integrate"
      write (u, "(A)")

      call mci%integrate (mci_instance, sampler, 1, 1000, results)
      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Display results"
      write (u, "(A)")

      call results%write (u)

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_base_7"

    end subroutine mci_base_7
```

Timer

Simple checks for the embedded timer.

⟨*MCI base: execute tests*⟩+≡
```
    call test (mci_base_8, "mci_base_8", &
         "timer", &
         u, results)
```

⟨*MCI base: tests*⟩+≡
```
    subroutine mci_base_8 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci

      write (u, "(A)")  "* Test output: mci_base_8"
      write (u, "(A)")  "*   Purpose: check timer availability"
```

```
   write (u, "(A)")

   write (u, "(A)")  "* Initialize integrator with timer"
   write (u, "(A)")

   allocate (mci_test_t :: mci)
   call mci%set_dimensions (2, 2)
   select type (mci)
   type is (mci_test_t)
      call mci%set_divisions (10)
   end select

   call mci%set_timer (active = .true.)
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Start timer"
   write (u, "(A)")

   call mci%start_timer ()
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Stop timer"
   write (u, "(A)")

   call mci%stop_timer ()
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Readout"
   write (u, "(A)")

   write (u, "(1x,A,F6.3)")  "Time = ", mci%get_time ()

   write (u, "(A)")
   write (u, "(A)")  "* Deactivate timer"
   write (u, "(A)")

   call mci%set_timer (active = .false.)
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_base_8"

 end subroutine mci_base_8
```

13.2 Simple midpoint integration

This is a most simple implementation of an integrator. The algorithm is the straightforward multi-dimensional midpoint rule, i.e., the integration hypercube is binned uniformly, the integrand is evaluated at the midpoints of each bin, and the result is the average. The binning is equivalent for all integration dimensions.

This rule is accurate to the order h^2, where h is the bin width. Given that $h = N^{-1/d}$, where d is the integration dimension and N is the total number of sampling points, we get a relative error of order $N^{-2/d}$. This is superior to MC integration if $d < 4$, and equivalent if $d = 4$. It is not worse than higher-order formulas (such as Gauss integration) if the integrand is not smooth, e.g., if it contains cuts.

The integrator is specifically single-channel. However, we do not limit the dimension.

⟨`mci_midpoint.f90`⟩≡

  ⟨*File header*⟩

```
module mci_midpoint

  use kinds !NODEP!
```
⟨*Use strings*⟩
⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use unit_tests

  use phs_base
  use rng_base
  use mci_base
```

  ⟨*Standard module head*⟩

  ⟨*MCI midpoint: public*⟩

  ⟨*MCI midpoint: types*⟩

  ⟨*MCI midpoint: test types*⟩

```
contains
```

  ⟨*MCI midpoint: procedures*⟩

  ⟨*MCI midpoint: tests*⟩

```
end module mci_midpoint
```

13.2.1 Integrator

The object contains the methods for integration and event generation. For the actual work and data storage, it spawns an instance object.

After an integration pass, we update the `max` parameter to indicate the maximum absolute value of the integrand that the integrator encountered. This

is required for event generation.

⟨MCI midpoint: public⟩≡
```
    public :: mci_midpoint_t
```

⟨MCI midpoint: types⟩≡
```
    type, extends (mci_t) :: mci_midpoint_t
       integer :: n_dim_binned = 0
       logical, dimension(:), allocatable :: dim_is_binned
       logical :: calls_known = .false.
       integer :: n_calls = 0
       integer :: n_calls_pos = 0
       integer :: n_calls_nul = 0
       integer :: n_calls_neg = 0
       real(default) :: integral_pos = 0
       real(default) :: integral_neg = 0
       integer, dimension(:), allocatable :: n_bin
       logical :: max_known = .false.
       real(default) :: max = 0
       real(default) :: min = 0
       real(default) :: max_abs = 0
       real(default) :: min_abs = 0
     contains
     ⟨MCI midpoint: mci midpoint: TBP⟩
  end type mci_midpoint_t
```

Finalizer: base version is sufficient

⟨MCI midpoint: mci midpoint: TBP⟩≡
```
    procedure :: final => mci_midpoint_final
```

⟨MCI midpoint: procedures⟩≡
```
    subroutine mci_midpoint_final (object)
      class(mci_midpoint_t), intent(inout) :: object
      call object%base_final ()
    end subroutine mci_midpoint_final
```

Output.

⟨MCI midpoint: mci midpoint: TBP⟩+≡
```
    procedure :: write => mci_midpoint_write
```

⟨MCI midpoint: procedures⟩+≡
```
    subroutine mci_midpoint_write (object, unit, pacify, md5sum_version)
      class(mci_midpoint_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      logical, intent(in), optional :: md5sum_version
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)") "Single-channel midpoint rule integrator:"
      call object%base_write (u, pacify, md5sum_version)
      if (object%n_dim_binned < object%n_dim) then
         write (u, "(3x,A,99(1x,I0))") "Flat dimensions      =", &
              pack ([(i, i = 1, object%n_dim)], mask = .not. object%dim_is_binned)
         write (u, "(3x,A,I0)")  "Number of binned dim = ", object%n_dim_binned
      end if
```

1624

```
      if (object%calls_known) then
         write (u, "(3x,A,99(1x,I0))")  "Number of bins       =", object%n_bin
         write (u, "(3x,A,I0)")  "Number of calls      = ", object%n_calls
         if (object%n_calls_pos /= object%n_calls) then
            write (u, "(3x,A,I0)")  "  positive value     = ", object%n_calls_pos
            write (u, "(3x,A,I0)")  "  zero value         = ", object%n_calls_nul
            write (u, "(3x,A,I0)")  "  negative value     = ", object%n_calls_neg
            write (u, "(3x,A,ES17.10)") &
                 "Integral (pos. part) = ", object%integral_pos
            write (u, "(3x,A,ES17.10)") &
                 "Integral (neg. part) = ", object%integral_neg
         end if
      end if
      if (object%max_known) then
         write (u, "(3x,A,ES17.10)")  "Maximum of integrand = ", object%max
         write (u, "(3x,A,ES17.10)")  "Minimum of integrand = ", object%min
         if (object%min /= object%min_abs) then
            write (u, "(3x,A,ES17.10)")  "Maximum (abs. value) = ", object%max_abs
            write (u, "(3x,A,ES17.10)")  "Minimum (abs. value) = ", object%min_abs
         end if
      end if
      if (allocated (object%rng))  call object%rng%write (u)
    end subroutine mci_midpoint_write
```

Startup message: short version.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: startup_message => mci_midpoint_startup_message
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_startup_message (mci, unit, n_calls)
      class(mci_midpoint_t), intent(in) :: mci
      integer, intent(in), optional :: unit, n_calls
      call mci%base_startup_message (unit = unit, n_calls = n_calls)
      if (mci%n_dim_binned < mci%n_dim) then
         write (msg_buffer, "(A,2(1x,I0,1x,A))") &
              "Integrator: Midpoint rule:", &
              mci%n_dim_binned, "binned dimensions"
      else
         write (msg_buffer, "(A,2(1x,I0,1x,A))") &
              "Integrator: Midpoint rule"
      end if
      call msg_message (unit = unit)
    end subroutine mci_midpoint_startup_message
```

The number of channels must be one.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: set_dimensions => mci_midpoint_set_dimensions
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_set_dimensions (mci, n_dim, n_channel)
      class(mci_midpoint_t), intent(inout) :: mci
      integer, intent(in) :: n_dim
      integer, intent(in) :: n_channel
      if (n_channel == 1) then
```

```
            mci%n_channel = n_channel
            mci%n_dim = n_dim
            allocate (mci%dim_is_binned (mci%n_dim))
            mci%dim_is_binned = .true.
            mci%n_dim_binned = count (mci%dim_is_binned)
            allocate (mci%n_bin (mci%n_dim))
            mci%n_bin = 0
         else
            call msg_fatal ("Attempt to initialize single-channel integrator &
                 &for multiple channels")
         end if
    end subroutine mci_midpoint_set_dimensions
```

Declare particular dimensions as flat. These dimensions will not be binned.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: declare_flat_dimensions => mci_midpoint_declare_flat_dimensions
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_declare_flat_dimensions (mci, dim_flat)
      class(mci_midpoint_t), intent(inout) :: mci
      integer, dimension(:), intent(in) :: dim_flat
      integer :: d
      mci%n_dim_binned = mci%n_dim - size (dim_flat)
      do d = 1, size (dim_flat)
         mci%dim_is_binned(dim_flat(d)) = .false.
      end do
      mci%n_dim_binned = count (mci%dim_is_binned)
    end subroutine mci_midpoint_declare_flat_dimensions
```

Declare particular channels as equivalent. This has no effect.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: declare_equivalences => mci_midpoint_ignore_equivalences
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_ignore_equivalences (mci, channel, dim_offset)
      class(mci_midpoint_t), intent(inout) :: mci
      type(phs_channel_t), dimension(:), intent(in) :: channel
      integer, intent(in) :: dim_offset
    end subroutine mci_midpoint_ignore_equivalences
```

Allocate instance with matching type.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: allocate_instance => mci_midpoint_allocate_instance
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_allocate_instance (mci, mci_instance)
      class(mci_midpoint_t), intent(in) :: mci
      class(mci_instance_t), intent(out), pointer :: mci_instance
      allocate (mci_midpoint_instance_t :: mci_instance)
    end subroutine mci_midpoint_allocate_instance
```

Integrate. The number of dimensions is arbitrary. We make sure that the number of calls is evenly distributed among the dimensions. The actual number of calls will typically be smaller than the requested number, but never smaller than 1.

The sampling over a variable number of dimensions implies a variable number of nested loops. We implement this by a recursive subroutine, one loop in each recursion level.

The number of iterations `n_it` is ignored. Also, the error is set to zero in the current implementation.

With this integrator, we allow the calculation to abort immediately when forced by a signal. There is no state that we can save, hence we do not catch an interrupt.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: integrate => mci_midpoint_integrate
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_integrate (mci, instance, sampler, n_it, n_calls, &
        results, pacify)
      class(mci_midpoint_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      integer, intent(in) :: n_it
      integer, intent(in) :: n_calls
      logical, intent(in), optional :: pacify
      class(mci_results_t), intent(inout), optional :: results
      real(default), dimension(:), allocatable :: x
      real(default) :: integral, integral_pos, integral_neg
      integer :: n_bin
      select type (instance)
      type is (mci_midpoint_instance_t)
         allocate (x (mci%n_dim))
         integral = 0
         integral_pos = 0
         integral_neg = 0
         select case (mci%n_dim_binned)
         case (1)
            n_bin = n_calls
         case (2:)
            n_bin = max (int (n_calls ** (1. / mci%n_dim_binned)), 1)
         end select
         where (mci%dim_is_binned)
            mci%n_bin = n_bin
         elsewhere
            mci%n_bin = 1
         end where
         mci%n_calls = product (mci%n_bin)
         mci%n_calls_pos = 0
         mci%n_calls_nul = 0
         mci%n_calls_neg = 0
         mci%calls_known = .true.
         call sample_dim (mci%n_dim)
         mci%integral = integral / mci%n_calls
         mci%integral_pos = integral_pos / mci%n_calls
```

1627

```
            mci%integral_neg = integral_neg / mci%n_calls
            mci%integral_known = .true.
            call instance%set_max ()
            if (present (results)) then
               call results%record (1, mci%n_calls, &
                    mci%integral, mci%error, mci%efficiency)
            end if
         end select
      contains
        recursive subroutine sample_dim (d)
          integer, intent(in) :: d
          integer :: i
          real(default) :: value
          do i = 1, mci%n_bin(d)
             x(d) = (i - 0.5_default) / mci%n_bin(d)
             if (d > 1) then
                call sample_dim (d - 1)
             else
                if (signal_is_pending ())  return
                call instance%evaluate (sampler, 1, x)
                value = instance%get_value ()
                if (value > 0) then
                   mci%n_calls_pos = mci%n_calls_pos + 1
                   integral = integral + value
                   integral_pos = integral_pos + value
                else if (value == 0) then
                   mci%n_calls_nul = mci%n_calls_nul + 1
                else
                   mci%n_calls_neg = mci%n_calls_neg + 1
                   integral = integral + value
                   integral_neg = integral_neg + value
                end if
             end if
          end do
        end subroutine sample_dim
      end subroutine mci_midpoint_integrate
```

Simulation initializer and finalizer: nothing to do here.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: prepare_simulation => mci_midpoint_ignore_prepare_simulation
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_ignore_prepare_simulation (mci)
      class(mci_midpoint_t), intent(inout) :: mci
    end subroutine mci_midpoint_ignore_prepare_simulation
```

Generate weighted event.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
    procedure :: generate_weighted_event => mci_midpoint_generate_weighted_event
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_generate_weighted_event (mci, instance, sampler)
      class(mci_midpoint_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout), target :: instance
```

```
      class(mci_sampler_t), intent(inout), target :: sampler
      real(default), dimension(mci%n_dim) :: x
      select type (instance)
      type is (mci_midpoint_instance_t)
         call mci%rng%generate (x)
         call instance%evaluate (sampler, 1, x)
         instance%excess_weight = 0
      end select
   end subroutine mci_midpoint_generate_weighted_event
```

For unweighted events, we generate weighted events and apply a simple rejection
step to the relative event weight, until an event passes.

   Note that we use the `max_abs` value stored in the configuration record, not
the one stored in the instance. The latter may change during event generation.
After an event generation pass is over, we may update the value for a subsequent
pass.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡
```
   procedure :: generate_unweighted_event => &
        mci_midpoint_generate_unweighted_event
```

⟨*MCI midpoint: procedures*⟩+≡
```
   subroutine mci_midpoint_generate_unweighted_event (mci, instance, sampler)
     class(mci_midpoint_t), intent(inout) :: mci
     class(mci_instance_t), intent(inout), target :: instance
     class(mci_sampler_t), intent(inout), target :: sampler
     real(default) :: x, norm, int
     select type (instance)
     type is (mci_midpoint_instance_t)
        if (mci%max_known .and. mci%max_abs > 0) then
           norm = abs (mci%max_abs * instance%safety_factor)
           REJECTION: do
              call mci%generate_weighted_event (instance, sampler)
              if (sampler%is_valid ()) then
                 call mci%rng%generate (x)
                 int = abs (instance%integrand)
                 if (x * norm <= int) then
                    if (norm > 0 .and. norm < int) then
                       instance%excess_weight = int / norm - 1
                    end if
                    exit REJECTION
                 end if
              end if
              if (signal_is_pending ())  return
           end do REJECTION
        else
           call msg_fatal ("Unweighted event generation: &
                &maximum of integrand is zero or unknown")
        end if
     end select
   end subroutine mci_midpoint_generate_unweighted_event
```

Rebuild an event, using the `state` input.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
      procedure :: rebuild_event => mci_midpoint_rebuild_event
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_rebuild_event (mci, instance, sampler, state)
      class(mci_midpoint_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      class(mci_state_t), intent(in) :: state
      select type (instance)
      type is (mci_midpoint_instance_t)
         call instance%recall (sampler, state)
      end select
    end subroutine mci_midpoint_rebuild_event
```

13.2.2 Integrator instance

Covering the case of flat dimensions, we store a complete x array. This is filled
when generating events.

⟨*MCI midpoint: public*⟩+≡
```
    public :: mci_midpoint_instance_t
```

⟨*MCI midpoint: types*⟩+≡
```
    type, extends (mci_instance_t) :: mci_midpoint_instance_t
      type(mci_midpoint_t), pointer :: mci => null ()
      logical :: max_known = .false.
      real(default) :: max = 0
      real(default) :: min = 0
      real(default) :: max_abs = 0
      real(default) :: min_abs = 0
      real(default) :: safety_factor = 1
      real(default) :: excess_weight = 0
    contains
    ⟨MCI midpoint: mci midpoint instance: TBP⟩
    end type mci_midpoint_instance_t
```

Output.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩≡
```
    procedure :: write => mci_midpoint_instance_write
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_instance_write (object, unit, pacify)
      class(mci_midpoint_instance_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A,9(1x,F12.10))")  "x =", object%x(:,1)
      write (u, "(1x,A,ES19.12)") "Integrand = ", object%integrand
      write (u, "(1x,A,ES19.12)") "Weight    = ", object%mci_weight
      if (object%safety_factor /= 1) then
         write (u, "(1x,A,ES19.12)") "Safety f  = ", object%safety_factor
      end if
      if (object%excess_weight /= 0) then
```

```
         write (u, "(1x,A,ES19.12)") "Excess    = ", object%excess_weight
      end if
      if (object%max_known) then
         write (u, "(1x,A,ES19.12)") "Maximum   = ", object%max
         write (u, "(1x,A,ES19.12)") "Minimum   = ", object%min
         if (object%min /= object%min_abs) then
            write (u, "(1x,A,ES19.12)") "Max.(abs) = ", object%max_abs
            write (u, "(1x,A,ES19.12)") "Min.(abs) = ", object%min_abs
         end if
      end if
   end subroutine mci_midpoint_instance_write
```

The finalizer is empty.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
   procedure :: final => mci_midpoint_instance_final
```

⟨*MCI midpoint: tests*⟩≡
```
   subroutine mci_midpoint_instance_final (object)
     class(mci_midpoint_instance_t), intent(inout) :: object
   end subroutine mci_midpoint_instance_final
```

Initializer.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
   procedure :: init => mci_midpoint_instance_init
```

⟨*MCI midpoint: procedures*⟩+≡
```
   subroutine mci_midpoint_instance_init (mci_instance, mci)
     class(mci_midpoint_instance_t), intent(out) :: mci_instance
     class(mci_t), intent(in), target :: mci
     call mci_instance%base_init (mci)
     select type (mci)
     type is (mci_midpoint_t)
        mci_instance%mci => mci
        call mci_instance%get_max ()
        mci_instance%selected_channel = 1
     end select
   end subroutine mci_midpoint_instance_init
```

Copy the stored extrema of the integrand in the instance record.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
   procedure :: get_max => mci_midpoint_instance_get_max
```

⟨*MCI midpoint: procedures*⟩+≡
```
   subroutine mci_midpoint_instance_get_max (instance)
     class(mci_midpoint_instance_t), intent(inout) :: instance
     associate (mci => instance%mci)
       if (mci%max_known) then
          instance%max_known = .true.
          instance%max = mci%max
          instance%min = mci%min
          instance%max_abs = mci%max_abs
          instance%min_abs = mci%min_abs
       end if
     end associate
```

```
        end subroutine mci_midpoint_instance_get_max
```

Reverse operations: recall the extrema, but only if they are wider than the extrema already stored in the configuration. Also recalculate the efficiency value.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
    procedure :: set_max => mci_midpoint_instance_set_max
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_instance_set_max (instance)
      class(mci_midpoint_instance_t), intent(inout) :: instance
      associate (mci => instance%mci)
        if (instance%max_known) then
           if (mci%max_known) then
              mci%max = max (mci%max, instance%max)
              mci%min = min (mci%min, instance%min)
              mci%max_abs = max (mci%max_abs, instance%max_abs)
              mci%min_abs = min (mci%min_abs, instance%min_abs)
           else
              mci%max = instance%max
              mci%min = instance%min
              mci%max_abs = instance%max_abs
              mci%min_abs = instance%min_abs
              mci%max_known = .true.
           end if
           if (mci%max_abs /= 0) then
              if (mci%integral == mci%integral_pos) then
                 mci%efficiency = mci%integral / mci%max_abs
                 mci%efficiency_known = .true.
              else if (mci%n_calls /= 0) then
                 mci%efficiency = &
                      (mci%n_calls_pos * mci%integral_pos &
                      - mci%n_calls_neg * mci%integral_neg) &
                      / mci%n_calls / mci%max_abs
                 mci%efficiency_known = .true.
              end if
           end if
        end if
      end associate
    end subroutine mci_midpoint_instance_set_max
```

The weight is the Jacobian of the mapping for the only channel.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
    procedure :: compute_weight => mci_midpoint_instance_compute_weight
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine mci_midpoint_instance_compute_weight (mci, c)
      class(mci_midpoint_instance_t), intent(inout) :: mci
      integer, intent(in) :: c
      select case (c)
      case (1)
         mci%mci_weight = mci%f(1)
      case default
         call msg_fatal ("MCI midpoint integrator: only single channel supported")
```

```
      end select
   end subroutine mci_midpoint_instance_compute_weight
```

Record the integrand. Update stored values for maximum and minimum.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
   procedure :: record_integrand => mci_midpoint_instance_record_integrand
```

⟨*MCI midpoint: procedures*⟩+≡
```
   subroutine mci_midpoint_instance_record_integrand (mci, integrand)
     class(mci_midpoint_instance_t), intent(inout) :: mci
     real(default), intent(in) :: integrand
     mci%integrand = integrand
     if (mci%max_known) then
        mci%max = max (mci%max, integrand)
        mci%min = min (mci%min, integrand)
        mci%max_abs = max (mci%max_abs, abs (integrand))
        mci%min_abs = min (mci%min_abs, abs (integrand))
     else
        mci%max = integrand
        mci%min = integrand
        mci%max_abs = abs (integrand)
        mci%min_abs = abs (integrand)
        mci%max_known = .true.
     end if
   end subroutine mci_midpoint_instance_record_integrand
```

We store the safety factor, otherwise nothing to do here.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
   procedure :: init_simulation => mci_midpoint_instance_init_simulation
   procedure :: final_simulation => mci_midpoint_instance_final_simulation
```

⟨*MCI midpoint: procedures*⟩+≡
```
   subroutine mci_midpoint_instance_init_simulation (instance, safety_factor)
     class(mci_midpoint_instance_t), intent(inout) :: instance
     real(default), intent(in), optional :: safety_factor
     if (present (safety_factor))  instance%safety_factor = safety_factor
   end subroutine mci_midpoint_instance_init_simulation

   subroutine mci_midpoint_instance_final_simulation (instance)
     class(mci_midpoint_instance_t), intent(inout) :: instance
   end subroutine mci_midpoint_instance_final_simulation
```

Return excess weight for the current event.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
   procedure :: get_event_excess => mci_midpoint_instance_get_event_excess
```

⟨*MCI midpoint: procedures*⟩+≡
```
   function mci_midpoint_instance_get_event_excess (mci) result (excess)
     class(mci_midpoint_instance_t), intent(in) :: mci
     real(default) :: excess
     excess = mci%excess_weight
   end function mci_midpoint_instance_get_event_excess
```

13.2.3 Unit tests

⟨*MCI midpoint: public*⟩+≡
```
    public :: mci_midpoint_test
```

⟨*MCI midpoint: tests*⟩+≡
```
    subroutine mci_midpoint_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨MCI midpoint: execute tests⟩
    end subroutine mci_midpoint_test
```


Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

This is the function $f(x) = 3x^2$ with integral $\int_0^1 f(x)\,dx = 1$ and maximum $f(1) = 3$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

Mimicking the behavior of a process object, we store the argument and result inside the sampler, so we can `fetch` results.

⟨*MCI midpoint: test types*⟩≡
```
    type, extends (mci_sampler_t) :: test_sampler_1_t
       real(default), dimension(:), allocatable :: x
       real(default) :: val
     contains
     ⟨MCI midpoint: test sampler 1: TBP⟩
    end type test_sampler_1_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI midpoint: test sampler 1: TBP*⟩≡
```
    procedure :: write => test_sampler_1_write
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_1_write (object, unit, testflag)
      class(test_sampler_1_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
    end subroutine test_sampler_1_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡
```
    procedure :: evaluate => test_sampler_1_evaluate
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
      class(test_sampler_1_t), intent(inout) :: sampler
      integer, intent(in) :: c
```

1634

```
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      if (allocated (sampler%x))  deallocate (sampler%x)
      allocate (sampler%x (size (x_in)))
      sampler%x = x_in
      sampler%val = 3 * x_in(1) ** 2
      call sampler%fetch (val, x, f)
    end subroutine test_sampler_1_evaluate
```

The point is always valid.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡
```
    procedure :: is_valid => test_sampler_1_is_valid
```

⟨*MCI midpoint: tests*⟩+≡
```
    function test_sampler_1_is_valid (sampler) result (valid)
      class(test_sampler_1_t), intent(in) :: sampler
      logical :: valid
      valid = .true.
    end function test_sampler_1_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡
```
    procedure :: rebuild => test_sampler_1_rebuild
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
      class(test_sampler_1_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      if (allocated (sampler%x))  deallocate (sampler%x)
      allocate (sampler%x (size (x_in)))
      sampler%x = x_in
      sampler%val = val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_1_rebuild
```

Extract the results.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡
```
    procedure :: fetch => test_sampler_1_fetch
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_1_fetch (sampler, val, x, f)
      class(test_sampler_1_t), intent(in) :: sampler
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      val = sampler%val
      x(:,1) = sampler%x
```

```
      f = 1
    end subroutine test_sampler_1_fetch
```

This is the function $f(x) = 3x^2 + 2y$ with integral $\int_0^1 f(x, y)\, dx\, dy = 2$ and maximum $f(1) = 5$.

⟨*MCI midpoint: test types*⟩+≡
```
    type, extends (mci_sampler_t) :: test_sampler_2_t
      real(default) :: val
      real(default), dimension(2) :: x
     contains
     ⟨MCI midpoint: test sampler 2: TBP⟩
    end type test_sampler_2_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI midpoint: test sampler 2: TBP*⟩≡
```
    procedure :: write => test_sampler_2_write
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_2_write (object, unit, testflag)
      class(test_sampler_2_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2 + 2 y"
    end subroutine test_sampler_2_write
```

Evaluate: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡
```
    procedure :: evaluate => test_sampler_2_evaluate
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
      class(test_sampler_2_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      sampler%x = x_in
      sampler%val = 3 * x_in(1) ** 2 + 2 * x_in(2)
      call sampler%fetch (val, x, f)
    end subroutine test_sampler_2_evaluate
```

The point is always valid.

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡
```
    procedure :: is_valid => test_sampler_2_is_valid
```

⟨*MCI midpoint: tests*⟩+≡
```
    function test_sampler_2_is_valid (sampler) result (valid)
      class(test_sampler_2_t), intent(in) :: sampler
      logical :: valid
```

```
      valid = .true.
    end function test_sampler_2_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡
```
    procedure :: rebuild => test_sampler_2_rebuild
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
      class(test_sampler_2_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      sampler%x = x_in
      sampler%val = val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_2_rebuild
```

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡
```
    procedure :: fetch => test_sampler_2_fetch
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_2_fetch (sampler, val, x, f)
      class(test_sampler_2_t), intent(in) :: sampler
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      val = sampler%val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_2_fetch
```

This is the function $f(x) = (1 - 3x^2)\,\theta(x - 1/2)$ with integral $\int_0^1 f(x)\,dx = -3/8$, minimum $f(1) = -2$ and maximum $f(1/2) = 1/4$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

⟨*MCI midpoint: test types*⟩+≡
```
    type, extends (mci_sampler_t) :: test_sampler_4_t
      real(default) :: val
      real(default), dimension(:), allocatable :: x
     contains
     ⟨MCI midpoint: test sampler 4: TBP⟩
    end type test_sampler_4_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI midpoint: test sampler 4: TBP*⟩≡
```
    procedure :: write => test_sampler_4_write
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_4_write (object, unit, testflag)
      class(test_sampler_4_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)") "Test sampler: f(x) = 1 - 3 x^2"
    end subroutine test_sampler_4_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡
```
    procedure :: evaluate => test_sampler_4_evaluate
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_4_evaluate (sampler, c, x_in, val, x, f)
      class(test_sampler_4_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      if (x_in(1) >= .5_default) then
         sampler%val = 1 - 3 * x_in(1) ** 2
      else
         sampler%val = 0
      end if
      if (.not. allocated (sampler%x))  allocate (sampler%x (size (x_in)))
      sampler%x = x_in
      call sampler%fetch (val, x, f)
    end subroutine test_sampler_4_evaluate
```

The point is always valid.

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡
```
    procedure :: is_valid => test_sampler_4_is_valid
```

⟨*MCI midpoint: tests*⟩+≡
```
    function test_sampler_4_is_valid (sampler) result (valid)
      class(test_sampler_4_t), intent(in) :: sampler
      logical :: valid
      valid = .true.
    end function test_sampler_4_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡
```
    procedure :: rebuild => test_sampler_4_rebuild
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_4_rebuild (sampler, c, x_in, val, x, f)
      class(test_sampler_4_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
```

```
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      sampler%x = x_in
      sampler%val = val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_4_rebuild
```

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡
```
    procedure :: fetch => test_sampler_4_fetch
```

⟨*MCI midpoint: procedures*⟩+≡
```
    subroutine test_sampler_4_fetch (sampler, val, x, f)
      class(test_sampler_4_t), intent(in) :: sampler
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      val = sampler%val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_4_fetch
```

One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

⟨*MCI midpoint: execute tests*⟩≡
```
    call test (mci_midpoint_1, "mci_midpoint_1", &
         "one-dimensional integral", &
         u, results)
```

⟨*MCI midpoint: tests*⟩+≡
```
    subroutine mci_midpoint_1 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler

      write (u, "(A)")  "* Test output: mci_midpoint_1"
      write (u, "(A)")  "*   Purpose: integrate function in one dimension"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_midpoint_t :: mci)
      call mci%set_dimensions (1, 1)

      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize instance"
      write (u, "(A)")
```

```
        call mci%allocate_instance (mci_instance)
        call mci_instance%init (mci)

        write (u, "(A)")  "* Initialize test sampler"
        write (u, "(A)")

        allocate (test_sampler_1_t :: sampler)
        call sampler%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Evaluate for x = 0.8"
        write (u, "(A)")

        call mci_instance%evaluate (sampler, 1, [0.8_default])
        call mci_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Evaluate for x = 0.7"
        write (u, "(A)")

        call mci_instance%evaluate (sampler, 1, [0.7_default])
        call mci_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Evaluate for x = 0.9"
        write (u, "(A)")

        call mci_instance%evaluate (sampler, 1, [0.9_default])
        call mci_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Integrate with n_calls = 1000"
        write (u, "(A)")

        call mci%integrate (mci_instance, sampler, 1, 1000)
        call mci%write (u)

        call mci_instance%final ()
        call mci%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: mci_midpoint_1"

    end subroutine mci_midpoint_1
```

Two-dimensional integration

Construct an integrator and use it for a two-dimensional sampler.

⟨*MCI midpoint: execute tests*⟩+≡
```
    call test (mci_midpoint_2, "mci_midpoint_2", &
        "two-dimensional integral", &
        u, results)
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine mci_midpoint_2 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler

  write (u, "(A)")  "* Test output: mci_midpoint_2"
  write (u, "(A)")  "*   Purpose: integrate function in two dimensions"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
  call mci%set_dimensions (2, 1)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_2_t :: sampler)
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for x = 0.8, y = 0.2"
  write (u, "(A)")

  call mci_instance%evaluate (sampler, 1, [0.8_default, 0.2_default])
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_calls = 1000"
  write (u, "(A)")

  call mci%integrate (mci_instance, sampler, 1, 1000)
  call mci%write (u)

  call mci_instance%final ()
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_midpoint_2"

end subroutine mci_midpoint_2
```

Two-dimensional integration with flat dimension

Construct an integrator and use it for a two-dimensional sampler, where the function is constant in the second dimension.

⟨*MCI midpoint: execute tests*⟩+≡
```
    call test (mci_midpoint_3, "mci_midpoint_3", &
        "two-dimensional integral with flat dimension", &
        u, results)
```

⟨*MCI midpoint: tests*⟩+≡
```
    subroutine mci_midpoint_3 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler

      write (u, "(A)")  "* Test output: mci_midpoint_3"
      write (u, "(A)")  "*   Purpose: integrate function with one flat dimension"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_midpoint_t :: mci)
      select type (mci)
      type is (mci_midpoint_t)
         call mci%set_dimensions (2, 1)
         call mci%declare_flat_dimensions ([2])
      end select

      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize instance"
      write (u, "(A)")

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      write (u, "(A)")  "* Initialize test sampler"
      write (u, "(A)")

      allocate (test_sampler_1_t :: sampler)
      call sampler%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Evaluate for x = 0.8, y = 0.2"
      write (u, "(A)")

      call mci_instance%evaluate (sampler, 1, [0.8_default, 0.2_default])
      call mci_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Integrate with n_calls = 1000"
      write (u, "(A)")
```

1642

```
      call mci%integrate (mci_instance, sampler, 1, 1000)
      call mci%write (u)

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_midpoint_3"

   end subroutine mci_midpoint_3
```

Integrand with sign flip

Construct an integrator and use it for a one-dimensional sampler.

⟨*MCI midpoint: execute tests*⟩+≡
```
      call test (mci_midpoint_4, "mci_midpoint_4", &
         "integrand with sign flip", &
         u, results)
```

⟨*MCI midpoint: tests*⟩+≡
```
   subroutine mci_midpoint_4 (u)
     integer, intent(in) :: u
     class(mci_t), allocatable, target :: mci
     class(mci_instance_t), pointer :: mci_instance => null ()
     class(mci_sampler_t), allocatable :: sampler

     write (u, "(A)")  "* Test output: mci_midpoint_4"
     write (u, "(A)")  "*   Purpose: integrate function with sign flip &
         &in one dimension"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize integrator"
     write (u, "(A)")

     allocate (mci_midpoint_t :: mci)
     call mci%set_dimensions (1, 1)

     call mci%write (u)

     write (u, "(A)")
     write (u, "(A)")  "* Initialize instance"
     write (u, "(A)")

     call mci%allocate_instance (mci_instance)
     call mci_instance%init (mci)

     write (u, "(A)")  "* Initialize test sampler"
     write (u, "(A)")

     allocate (test_sampler_4_t :: sampler)
     call sampler%write (u)
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Evaluate for x = 0.8"
      write (u, "(A)")

      call mci_instance%evaluate (sampler, 1, [0.8_default])
      call mci_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")   "* Integrate with n_calls = 1000"
      write (u, "(A)")

      call mci%integrate (mci_instance, sampler, 1, 1000)
      call mci%write (u)

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: mci_midpoint_4"

    end subroutine mci_midpoint_4
```

Weighted events

Generate weighted events. Without rejection, we do not need to know maxima and minima, so we can start generating events immediately. We have two dimensions.

⟨*MCI midpoint: execute tests*⟩+≡
```
    call test (mci_midpoint_5, "mci_midpoint_5", &
        "weighted events", &
        u, results)
```

⟨*MCI midpoint: tests*⟩+≡
```
    subroutine mci_midpoint_5 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng
      class(mci_state_t), allocatable :: state

      write (u, "(A)")   "* Test output: mci_midpoint_5"
      write (u, "(A)")   "*   Purpose: generate weighted events"
      write (u, "(A)")

      write (u, "(A)")   "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_midpoint_t :: mci)
      call mci%set_dimensions (2, 1)

      call mci%write (u)
```

```
write (u, "(A)")
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_2_t :: sampler)

write (u, "(A)")  "* Initialize random-number generator"
write (u, "(A)")

allocate (rng_test_t :: rng)
call rng%init ()
call mci%import_rng (rng)

write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

call mci%generate_weighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

call mci%generate_weighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Store data"
write (u, "(A)")

allocate (state)
call mci_instance%store (state)
call mci_instance%final ()
deallocate (mci_instance)

call state%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recall data and rebuild event"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)
call mci%rebuild_event (mci_instance, sampler, state)

call mci_instance%write (u)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    deallocate (mci_instance)
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_midpoint_5"

  end subroutine mci_midpoint_5
```

Unweighted events

Generate unweighted events. The integrand has a sign flip in it.

⟨*MCI midpoint: execute tests*⟩+≡
```
    call test (mci_midpoint_6, "mci_midpoint_6", &
         "unweighted events", &
         u, results)
```

⟨*MCI midpoint: tests*⟩+≡
```
  subroutine mci_midpoint_6 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_midpoint_6"
    write (u, "(A)")  "*   Purpose: generate unweighted events"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_midpoint_t :: mci)
    call mci%set_dimensions (1, 1)

    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_4_t :: sampler)

    write (u, "(A)")  "* Initialize random-number generator"
    write (u, "(A)")

    allocate (rng_test_t :: rng)
    call rng%init ()
```

```
        call mci%import_rng (rng)

        write (u, "(A)")  "* Integrate (determine maximum of integrand"
        write (u, "(A)")
        call mci%integrate (mci_instance, sampler, 1, 1000)
        call mci%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Generate unweighted event"
        write (u, "(A)")

        call mci%generate_unweighted_event (mci_instance, sampler)
        call mci_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call mci_instance%final ()
        deallocate (mci_instance)
        call mci%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: mci_midpoint_6"

      end subroutine mci_midpoint_6
```

Excess weight

Generate unweighted events. With only 2 points for integration, the maximum
of the integrand is too low, and we produce excess weight.

⟨*MCI midpoint: execute tests*⟩+≡
```
    call test (mci_midpoint_7, "mci_midpoint_7", &
        "excess weight", &
        u, results)
```

⟨*MCI midpoint: tests*⟩+≡
```
    subroutine mci_midpoint_7 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_midpoint_7"
      write (u, "(A)")  "*   Purpose: generate unweighted event &
          &with excess weight"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_midpoint_t :: mci)
      call mci%set_dimensions (1, 1)
```

1647

```
write (u, "(A)")  "* Initialize instance"
write (u, "(A)")

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")  "* Initialize test sampler"
write (u, "(A)")

allocate (test_sampler_4_t :: sampler)

write (u, "(A)")  "* Initialize random-number generator"
write (u, "(A)")

allocate (rng_test_t :: rng)
call rng%init ()
call mci%import_rng (rng)

write (u, "(A)")  "* Integrate (determine maximum of integrand"
write (u, "(A)")
call mci%integrate (mci_instance, sampler, 1, 2)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate unweighted event"
write (u, "(A)")

call mci_instance%init_simulation ()
call mci%generate_unweighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Use getter methods"
write (u, "(A)")

write (u, "(1x,A,1x,ES19.12)")  "weight =", mci_instance%get_event_weight ()
write (u, "(1x,A,1x,ES19.12)")  "excess =", mci_instance%get_event_excess ()

write (u, "(A)")
write (u, "(A)")  "* Apply safety factor"
write (u, "(A)")

call mci_instance%init_simulation (safety_factor = 2.1_default)

write (u, "(A)")  "* Generate unweighted event"
write (u, "(A)")

call mci%generate_unweighted_event (mci_instance, sampler)
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Use getter methods"
write (u, "(A)")
```

```
      write (u, "(1x,A,1x,ES19.12)")  "weight =", mci_instance%get_event_weight ()
      write (u, "(1x,A,1x,ES19.12)")  "excess =", mci_instance%get_event_excess ()

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call mci_instance%final ()
      deallocate (mci_instance)
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_midpoint_7"

    end subroutine mci_midpoint_7
```

13.3 VAMP interface

The standard method for integration is VAMP: the multi-channel version of the VEGAS algorithm. Each parameterization (channel) of the hypercube is binned in each dimension. The binning is equally equidistant, but an iteration of the integration procedure, the binning is updated for each dimension, according to the variance distribution of the integrand, summed over all other dimension. In the next iteration, the binning approximates (hopefully) follows the integrand more closely, and the accuracy of the result is increased. Furthermore, the relative weight of the individual channels is also updated after an iteration.

The bin distribution is denoted as the grid for a channel, which we can write to file and reuse later.

In our implementation we specify the generic VAMP algorithm more tightly: the number of bins is equal for all dimensions, the initial weights are all equal. The user controls whether to update bins and/or weights after each iteration. The integration is organized in passes, each one consisting of several iterations with a common number of calls to the integrand. The first passes are intended as warmup, so the results are displayed but otherwise discarded. In the final pass, the integration estimates for the individual iterations are averaged for the final result.

⟨mci_vamp.f90⟩≡
  ⟨File header⟩

```
  module mci_vamp

    use kinds !NODEP!
```
⟨Use strings⟩
⟨Use file utils⟩
```
    use limits, only: FMT_12, FMT_14, FMT_17, FMT_19 !NODEP!
    use diagnostics !NODEP!
    use constants !NODEP!
    use unit_tests
    use md5
```

```
      use phs_base
      use rng_base
      use rng_tao
      use mci_base

      use vamp !NODEP!
      use exceptions !NODEP!
```

⟨*Standard module head*⟩

⟨*MCI vamp: public*⟩

⟨*MCI vamp: types*⟩

⟨*MCI vamp: interfaces*⟩

⟨*MCI vamp: test types*⟩

```
contains
```

⟨*MCI vamp: procedures*⟩

⟨*MCI vamp: tests*⟩

```
      end module mci_vamp
```

13.3.1 Grid parameters

This is a transparent container. It holds the parameters that are stored in grid files, and are checked when grid files are read.

⟨*MCI vamp: public*⟩≡
```
      public :: grid_parameters_t
```

⟨*MCI vamp: types*⟩≡
```
      type :: grid_parameters_t
         integer :: threshold_calls = 0
         integer :: min_calls_per_channel = 10
         integer :: min_calls_per_bin = 10
         integer :: min_bins = 3
         integer :: max_bins = 20
         logical :: stratified = .true.
         logical :: use_vamp_equivalences = .true.
         real(default) :: channel_weights_power = 0.25_default
         real(default) :: accuracy_goal = 0
         real(default) :: error_goal = 0
         real(default) :: rel_error_goal = 0
       contains
       ⟨MCI vamp: grid parameters: TBP⟩
      end type grid_parameters_t
```

I/O:

⟨*MCI vamp: grid parameters: TBP*⟩≡
```
      procedure :: write => grid_parameters_write
```

⟨*MCI vamp: procedures*⟩≡
```
subroutine grid_parameters_write (object, unit)
  class(grid_parameters_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A,I0)") "threshold_calls     = ", &
       object%threshold_calls
  write (u, "(3x,A,I0)") "min_calls_per_channel = ", &
       object%min_calls_per_channel
  write (u, "(3x,A,I0)") "min_calls_per_bin    = ", &
       object%min_calls_per_bin
  write (u, "(3x,A,I0)") "min_bins          = ", &
       object%min_bins
  write (u, "(3x,A,I0)") "max_bins          = ", &
       object%max_bins
  write (u, "(3x,A,L1)") "stratified        = ", &
       object%stratified
  write (u, "(3x,A,L1)") "use_vamp_equivalences = ", &
       object%use_vamp_equivalences
  write (u, "(3x,A,F10.7)") "channel_weights_power = ", &
       object%channel_weights_power
  if (object%accuracy_goal > 0) then
     write (u, "(3x,A,F10.7)") "accuracy_goal      = ", &
          object%accuracy_goal
  end if
  if (object%error_goal > 0) then
     write (u, "(3x,A,F10.7)") "error_goal        = ", &
          object%error_goal
  end if
  if (object%rel_error_goal > 0) then
     write (u, "(3x,A,F10.7)") "rel_error_goal     = ", &
          object%rel_error_goal
  end if
end subroutine grid_parameters_write
```

13.3.2 History parameters

The history parameters are also stored in a transparent container. This is not a part of the grid definition, and should not be included in the MD5 sum.

⟨*MCI vamp: public*⟩+≡
```
public :: history_parameters_t
```

⟨*MCI vamp: types*⟩+≡
```
type :: history_parameters_t
  logical :: global = .true.
  logical :: global_verbose = .false.
  logical :: channel = .false.
  logical :: channel_verbose = .false.
 contains
```
⟨*MCI vamp: history parameters: TBP*⟩
```
end type history_parameters_t
```

1651

I/O:

⟨*MCI vamp: history parameters: TBP*⟩≡
```
    procedure :: write => history_parameters_write
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine history_parameters_write (object, unit)
      class(history_parameters_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A,L1)") "history(global)      = ", object%global
      write (u, "(3x,A,L1)") "history(global) verb. = ", object%global_verbose
      write (u, "(3x,A,L1)") "history(channels)    = ", object%channel
      write (u, "(3x,A,L1)") "history(chann.) verb. = ", object%channel_verbose
    end subroutine history_parameters_write
```

13.3.3   Integration pass

We store the parameters for each integration pass in a linked list.

⟨*MCI vamp: types*⟩+≡
```
    type :: pass_t
       integer :: i_pass = 0
       integer :: i_first_it = 0
       integer :: n_it = 0
       integer :: n_calls = 0
       integer :: n_bins = 0
       logical :: adapt_grids = .false.
       logical :: adapt_weights = .false.
       logical :: is_final_pass = .false.
       logical :: integral_defined = .false.
       integer, dimension(:), allocatable :: calls
       real(default), dimension(:), allocatable :: integral
       real(default), dimension(:), allocatable :: error
       real(default), dimension(:), allocatable :: efficiency
       type(vamp_history), dimension(:), allocatable :: v_history
       type(vamp_history), dimension(:,:), allocatable :: v_histories
       type(pass_t), pointer :: next => null ()
     contains
     ⟨MCI vamp: pass: TBP⟩
    end type pass_t
```

Finalizer. The VAMP histories contain a pointer array.

⟨*MCI vamp: pass: TBP*⟩≡
```
    procedure :: final => pass_final
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine pass_final (object)
      class(pass_t), intent(inout) :: object
      if (allocated (object%v_history)) then
         call vamp_delete_history (object%v_history)
      end if
      if (allocated (object%v_histories)) then
         call vamp_delete_history (object%v_histories)
```

```
         end if
      end subroutine pass_final
```

Output. Note that the precision of the numerical values should match the
precision for comparing output from file with data.

⟨*MCI vamp: pass: TBP*⟩+≡
```
      procedure :: write => pass_write
```

⟨*MCI vamp: procedures*⟩+≡
```
      subroutine pass_write (object, unit, pacify)
        class(pass_t), intent(in) :: object
        integer, intent(in) :: unit
        logical, intent(in), optional :: pacify
        integer :: u, i
        character(len=7) :: fmt
        call pac_fmt (fmt, FMT_17, FMT_14, pacify)
        u = output_unit (unit)
        write (u, "(3x,A,I0)")  "n_it          = ", object%n_it
        write (u, "(3x,A,I0)")  "n_calls       = ", object%n_calls
        write (u, "(3x,A,I0)")  "n_bins        = ", object%n_bins
        write (u, "(3x,A,L1)")  "adapt grids   = ", object%adapt_grids
        write (u, "(3x,A,L1)")  "adapt weights = ", object%adapt_weights
        if (object%integral_defined) then
           write (u, "(3x,A)")  "Results:  [it, calls, integral, error, efficiency]"
           do i = 1, object%n_it
              write (u, "(5x,I0,1x,I0,3(1x," // fmt // "))") &
                   i, object%calls(i), object%integral(i), object%error(i), &
                   object%efficiency(i)
           end do
        else
           write (u, "(3x,A)")  "Results: [undefined]"
        end if
      end subroutine pass_write
```

Read and reconstruct the pass.

⟨*MCI vamp: pass: TBP*⟩+≡
```
      procedure :: read => pass_read
```

⟨*MCI vamp: procedures*⟩+≡
```
      subroutine pass_read (object, u, n_pass, n_it)
        class(pass_t), intent(out) :: object
        integer, intent(in) :: u, n_pass, n_it
        integer :: i, j
        character(80) :: buffer
        object%i_pass = n_pass + 1
        object%i_first_it = n_it + 1
        call read_ival (u, object%n_it)
        call read_ival (u, object%n_calls)
        call read_ival (u, object%n_bins)
        call read_lval (u, object%adapt_grids)
        call read_lval (u, object%adapt_weights)
        allocate (object%calls (object%n_it), source = 0)
        allocate (object%integral (object%n_it), source = 0._default)
        allocate (object%error (object%n_it), source = 0._default)
```

```
      allocate (object%efficiency (object%n_it), source = 0._default)
      read (u, "(A)")  buffer
      select case (trim (adjustl (buffer)))
      case ("Results:  [it, calls, integral, error, efficiency]")
         do i = 1, object%n_it
            read (u, *) &
                 j, object%calls(i), object%integral(i), object%error(i), &
                 object%efficiency(i)
         end do
         object%integral_defined = .true.
      case ("Results: [undefined]")
         object%integral_defined = .false.
      case default
         call msg_fatal ("Reading integration pass: corrupted file")
      end select
    end subroutine pass_read
```

Write the VAMP history for this pass. (The subroutine writes the whole array at once.)

⟨*MCI vamp: pass: TBP*⟩+≡
```
    procedure :: write_history => pass_write_history
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine pass_write_history (pass, unit)
      class(pass_t), intent(in) :: pass
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (allocated (pass%v_history)) then
         call vamp_write_history (u, pass%v_history)
      else
         write (u, "(1x,A)")  "Global history: [undefined]"
      end if
      if (allocated (pass%v_histories)) then
         write (u, "(1x,A)")  "Channel histories:"
         call vamp_write_history (u, pass%v_histories)
      else
         write (u, "(1x,A)")  "Channel histories: [undefined]"
      end if
    end subroutine pass_write_history
```

Given a number of calls and iterations, compute remaining data.

⟨*MCI vamp: pass: TBP*⟩+≡
```
    procedure :: configure => pass_configure
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine pass_configure (pass, n_it, n_calls, min_calls, &
         min_bins, max_bins, min_channel_calls)
      class(pass_t), intent(inout) :: pass
      integer, intent(in) :: n_it, n_calls, min_channel_calls
      integer, intent(in) :: min_calls, min_bins, max_bins
      pass%n_it = n_it
      if (min_calls /= 0) then
         pass%n_bins =  max (min_bins, &
```

```
                  min (n_calls / min_calls, max_bins))
          else
             pass%n_bins = max_bins
          end if
          pass%n_calls = max (n_calls, max (min_calls, min_channel_calls))
          if (pass%n_calls /= n_calls) then
             write (msg_buffer, "(A,I0)")  "VAMP: too few calls, resetting " &
                   // "n_calls to ", pass%n_calls
             call msg_warning ()
          end if
          allocate (pass%calls (n_it), source = 0)
          allocate (pass%integral (n_it), source = 0._default)
          allocate (pass%error (n_it), source = 0._default)
          allocate (pass%efficiency (n_it), source = 0._default)
        end subroutine pass_configure
```

Allocate the VAMP history and give options. We assume that the `configure`
routine above has been executed, so the number of iterations is known.

⟨*MCI vamp: pass: TBP*⟩+≡
```
        procedure :: configure_history => pass_configure_history
```

⟨*MCI vamp: procedures*⟩+≡
```
        subroutine pass_configure_history (pass, n_channels, par)
          class(pass_t), intent(inout) :: pass
          integer, intent(in) :: n_channels
          type(history_parameters_t), intent(in) :: par
          if (par%global) then
             allocate (pass%v_history (pass%n_it))
             call vamp_create_history (pass%v_history, &
                   verbose = par%global_verbose)
          end if
          if (par%channel) then
             allocate (pass%v_histories (pass%n_it, n_channels))
             call vamp_create_history (pass%v_histories, &
                   verbose = par%channel_verbose)
          end if
        end subroutine pass_configure_history
```

Given two pass objects, compare them. All parameters must match. Where
integrations are done in both (number of calls nonzero), the results must be
equal (up to numerical noise).

   The allocated array sizes might be different, but should match up to the
common `n_it` value.

⟨*MCI vamp: interfaces*⟩≡
```
        interface operator (.matches.)
           module procedure pass_matches
        end interface operator (.matches.)
```

⟨*MCI vamp: procedures*⟩+≡
```
        function pass_matches (pass, ref) result (ok)
          type(pass_t), intent(in) :: pass, ref
          integer :: n
          logical :: ok
```

```
      ok = .true.
      if (ok)  ok = pass%i_pass == ref%i_pass
      if (ok)  ok = pass%i_first_it == ref%i_first_it
      if (ok)  ok = pass%n_it == ref%n_it
      if (ok)  ok = pass%n_calls == ref%n_calls
      if (ok)  ok = pass%n_bins == ref%n_bins
      if (ok)  ok = pass%adapt_grids .eqv. ref%adapt_grids
      if (ok)  ok = pass%adapt_weights .eqv. ref%adapt_weights
      if (ok)  ok = pass%integral_defined .eqv. ref%integral_defined
      if (pass%integral_defined) then
         n = pass%n_it
         if (ok)  ok = all (pass%calls(:n) == ref%calls(:n))
         if (ok)  ok = all (pass%integral(:n) .matches. ref%integral(:n))
         if (ok)  ok = all (pass%error(:n) .matches. ref%error(:n))
         if (ok)  ok = all (pass%efficiency(:n) .matches. ref%efficiency(:n))
      end if
    end function pass_matches
```

Update a pass object, given a reference. The parameters must match, except
for the `n_it` entry. The number of complete iterations must be less or equal to
the reference, and the number of complete iterations in the reference must be no
larger than `n_it`. Where results are present in both passes, they must match.
Where results are present in the reference only, the pass is updated accordingly.

⟨*MCI vamp: pass: TBP*⟩+≡
```
    procedure :: update => pass_update
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine pass_update (pass, ref, ok)
      class(pass_t), intent(inout) :: pass
      type(pass_t), intent(in) :: ref
      logical, intent(out) :: ok
      integer :: n, n_ref
      ok = .true.
      if (ok)  ok = pass%i_pass == ref%i_pass
      if (ok)  ok = pass%i_first_it == ref%i_first_it
      if (ok)  ok = pass%n_calls == ref%n_calls
      if (ok)  ok = pass%n_bins == ref%n_bins
      if (ok)  ok = pass%adapt_grids .eqv. ref%adapt_grids
      if (ok)  ok = pass%adapt_weights .eqv. ref%adapt_weights
      if (ok) then
         if (ref%integral_defined) then
            if (.not. allocated (pass%calls)) then
               allocate (pass%calls (pass%n_it), source = 0)
               allocate (pass%integral (pass%n_it), source = 0._default)
               allocate (pass%error (pass%n_it), source = 0._default)
               allocate (pass%efficiency (pass%n_it), source = 0._default)
            end if
            n = count (pass%calls /= 0)
            n_ref = count (ref%calls /= 0)
            ok = n <= n_ref .and. n_ref <= pass%n_it
            if (ok)  ok = all (pass%calls(:n) == ref%calls(:n))
            if (ok)  ok = all (pass%integral(:n) .matches. ref%integral(:n))
            if (ok)  ok = all (pass%error(:n) .matches. ref%error(:n))
            if (ok)  ok = all (pass%efficiency(:n) .matches. ref%efficiency(:n))
```

```
                if (ok) then
                    pass%calls(n+1:n_ref) = ref%calls(n+1:n_ref)
                    pass%integral(n+1:n_ref) = ref%integral(n+1:n_ref)
                    pass%error(n+1:n_ref) = ref%error(n+1:n_ref)
                    pass%efficiency(n+1:n_ref) = ref%efficiency(n+1:n_ref)
                    pass%integral_defined = any (pass%calls /= 0)
                end if
            end if
        end if
    end subroutine pass_update
```

Match two real numbers: they are equal up to a tolerance, which is 10^{-8}, matching the number of digits that are output by `pass_write`. In particular, if one number is exactly zero, the other one must also be zero.

⟨*MCI vamp: interfaces*⟩+≡
```
    interface operator (.matches.)
       module procedure real_matches
    end interface operator (.matches.)
```

⟨*MCI vamp: procedures*⟩+≡
```
    elemental function real_matches (x, y) result (ok)
      real(default), intent(in) :: x, y
      logical :: ok
      real(default), parameter :: tolerance = 1.e-8_default
      ok = abs (x - y) <= tolerance * max (abs (x), abs (y))
    end function real_matches
```

Return the index of the most recent complete integration. If there is none, return zero.

⟨*MCI vamp: pass: TBP*⟩+≡
```
    procedure :: get_integration_index => pass_get_integration_index
```

⟨*MCI vamp: procedures*⟩+≡
```
    function pass_get_integration_index (pass) result (n)
      class (pass_t), intent(in) :: pass
      integer :: n
      integer :: i
      n = 0
      if (allocated (pass%calls)) then
         do i = 1, pass%n_it
            if (pass%calls(i) == 0)  exit
            n = i
         end do
      end if
    end function pass_get_integration_index
```

Return the most recent integral and error, if available.

⟨*MCI vamp: pass: TBP*⟩+≡
```
    procedure :: get_calls => pass_get_calls
    procedure :: get_integral => pass_get_integral
    procedure :: get_error => pass_get_error
    procedure :: get_efficiency => pass_get_efficiency
```

⟨*MCI vamp: procedures*⟩+≡
```
function pass_get_calls (pass) result (calls)
  class(pass_t), intent(in) :: pass
  integer :: calls
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
     calls = pass%calls(n)
  else
     calls = 0
  end if
end function pass_get_calls

function pass_get_integral (pass) result (integral)
  class(pass_t), intent(in) :: pass
  real(default) :: integral
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
     integral = pass%integral(n)
  else
     integral = 0
  end if
end function pass_get_integral

function pass_get_error (pass) result (error)
  class(pass_t), intent(in) :: pass
  real(default) :: error
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
     error = pass%error(n)
  else
     error = 0
  end if
end function pass_get_error

function pass_get_efficiency (pass) result (efficiency)
  class(pass_t), intent(in) :: pass
  real(default) :: efficiency
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
     efficiency = pass%efficiency(n)
  else
     efficiency = 0
  end if
end function pass_get_efficiency
```

13.3.4 Integrator

⟨*MCI vamp: public*⟩+≡
```
public :: mci_vamp_t
```

1658

⟨*MCI vamp: types*⟩+≡
```
type, extends (mci_t) :: mci_vamp_t
   logical, dimension(:), allocatable :: dim_is_flat
   type(grid_parameters_t) :: grid_par
   type(history_parameters_t) :: history_par
   integer :: min_calls = 0
   type(pass_t), pointer :: first_pass => null ()
   type(pass_t), pointer :: current_pass => null ()
   type(vamp_equivalences_t) :: equivalences
   logical :: rebuild = .true.
   logical :: check_grid_file = .true.
   logical :: grid_filename_set = .false.
   logical :: negative_weights = .false.
   logical :: verbose = .false.
   type(string_t) :: grid_filename
   character(32) :: md5sum_adapted = ""
 contains
 ⟨MCI vamp: mci vamp: TBP⟩
end type mci_vamp_t
```

Reset: delete integration-pass entries.

⟨*MCI vamp: mci vamp: TBP*⟩≡
```
procedure :: reset => mci_vamp_reset
```

⟨*MCI vamp: procedures*⟩+≡
```
subroutine mci_vamp_reset (object)
  class(mci_vamp_t), intent(inout) :: object
  type(pass_t), pointer :: current_pass
  do while (associated (object%first_pass))
     current_pass => object%first_pass
     object%first_pass => current_pass%next
     call current_pass%final ()
     deallocate (current_pass)
  end do
  object%current_pass => null ()
end subroutine mci_vamp_reset
```

Finalizer: reset and finalize the equivalences list.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
procedure :: final => mci_vamp_final
```

⟨*MCI vamp: procedures*⟩+≡
```
subroutine mci_vamp_final (object)
  class(mci_vamp_t), intent(inout) :: object
  call object%reset ()
  call vamp_equivalences_final (object%equivalences)
  call object%base_final ()
end subroutine mci_vamp_final
```

Output. Do not output the grids themselves, this may result in tons of data.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
procedure :: write => mci_vamp_write
```

1659

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_write (object, unit, pacify, md5sum_version)
      class(mci_vamp_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      logical, intent(in), optional :: md5sum_version
      type(pass_t), pointer :: current_pass
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "VAMP integrator:"
      call object%base_write (u, pacify, md5sum_version)
      if (allocated (object%dim_is_flat)) then
         write (u, "(3x,A,999(1x,I0))")  "Flat dimensions   =", &
              pack ([(i, i = 1, object%n_dim)], object%dim_is_flat)
      end if
      write (u, "(1x,A)")  "Grid parameters:"
      call object%grid_par%write (u)
      write (u, "(3x,A,I0)") "min_calls             = ", object%min_calls
      write (u, "(3x,A,L1)") "negative weights      = ", &
           object%negative_weights
      write (u, "(3x,A,L1)") "verbose               = ", &
           object%verbose
      if (object%grid_par%use_vamp_equivalences) then
         call vamp_equivalences_write (object%equivalences, u)
      end if
      current_pass => object%first_pass
      do while (associated (current_pass))
         write (u, "(1x,A,I0,A)")  "Integration pass:"
         call current_pass%write (u, pacify)
         current_pass => current_pass%next
      end do
      if (object%md5sum_adapted /= "") then
         write (u, "(1x,A,A,A)")  "MD5 sum (including results) = '", &
              object%md5sum_adapted, "'"
      end if
    end subroutine mci_vamp_write
```

Write the history parameters.
⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: write_history_parameters => mci_vamp_write_history_parameters
```
⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_write_history_parameters (mci, unit)
      class(mci_vamp_t), intent(in) :: mci
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "VAMP history parameters:"
      call mci%history_par%write (unit)
    end subroutine mci_vamp_write_history_parameters
```

Write the history, iterating over passes. We keep this separate from the generic
`write` routine.
⟨*MCI vamp: mci vamp: TBP*⟩+≡

1660

```
      procedure :: write_history => mci_vamp_write_history
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_write_history (mci, unit)
      class(mci_vamp_t), intent(in) :: mci
      integer, intent(in), optional :: unit
      type(pass_t), pointer :: current_pass
      integer :: i_pass
      integer :: u
      u = output_unit (unit)
      if (associated (mci%first_pass)) then
         write (u, "(1x,A)")  "VAMP history (global):"
         i_pass = 0
         current_pass => mci%first_pass
         do while (associated (current_pass))
            i_pass = i_pass + 1
            write (u, "(1x,A,I0,':')")  "Pass #", i_pass
            call current_pass%write_history (u)
            current_pass => current_pass%next
         end do
      end if
    end subroutine mci_vamp_write_history
```

Compute the MD5 sum, including the configuration MD5 sum and the printout, which incorporates the current results.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: compute_md5sum => mci_vamp_compute_md5sum
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_compute_md5sum (mci, pacify)
      class(mci_vamp_t), intent(inout) :: mci
      logical, intent(in), optional :: pacify
      integer :: u
      mci%md5sum_adapted = ""
      u = free_unit ()
      open (u, status = "scratch", action = "readwrite")
      write (u, "(A)")  mci%md5sum
      call mci%write (u, pacify, md5sum_version = .true.)
      rewind (u)
      mci%md5sum_adapted = md5sum (u)
      close (u)
    end subroutine mci_vamp_compute_md5sum
```

Return the MD5 sum: If available, return the adapted one.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: get_md5sum => mci_vamp_get_md5sum
```

⟨*MCI vamp: procedures*⟩+≡
```
    function mci_vamp_get_md5sum (mci) result (md5sum)
      class(mci_vamp_t), intent(in) :: mci
      character(32) :: md5sum
      if (mci%md5sum_adapted /= "") then
         md5sum = mci%md5sum_adapted
      else
```

```
      md5sum = mci%md5sum
    end if
  end function mci_vamp_get_md5sum
```

Startup message: short version.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: startup_message => mci_vamp_startup_message
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_startup_message (mci, unit, n_calls)
      class(mci_vamp_t), intent(in) :: mci
      integer, intent(in), optional :: unit, n_calls
      integer :: num_calls, n_bins
      if (present (n_calls)) then
         num_calls = n_calls
      else
         num_calls = 0
      end if
      if (mci%min_calls /= 0) then
         n_bins =  max (mci%grid_par%min_bins, &
              min (num_calls / mci%min_calls, &
              mci%grid_par%max_bins))
      else
         n_bins = mci%grid_par%max_bins
      end if
      call mci%base_startup_message (unit = unit, n_calls = n_calls)
      if (mci%grid_par%use_vamp_equivalences) then
         write (msg_buffer, "(A,2(1x,I0,1x,A))") &
              "Integrator: Using VAMP channel equivalences"
         call msg_message (unit = unit)
      end if
      write (msg_buffer, "(A,2(1x,I0,1x,A),L1)") &
           "Integrator:", num_calls, &
           "initial calls,", n_bins, &
           "bins, stratified = ", &
           mci%grid_par%stratified
      call msg_message (unit = unit)
      write (msg_buffer, "(A,2(1x,I0,1x,A))") &
           "Integrator: VAMP"
      call msg_message (unit = unit)
    end subroutine mci_vamp_startup_message
```

Set the MCI index (necessary for processes with multiple components). We
append the index to the grid filename, just before the final dotted suffix.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: record_index => mci_vamp_record_index
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_record_index (mci, i_mci)
      class(mci_vamp_t), intent(inout) :: mci
      integer, intent(in) :: i_mci
      type(string_t) :: basename, suffix
      character(32) :: buffer
      if (mci%grid_filename_set) then
```

1662

```
      basename = mci%grid_filename
      call split (basename, suffix, ".", back=.true.)
      write (buffer, "(I0)")  i_mci
      if (basename /= "") then
         mci%grid_filename = basename // "_m" // trim (buffer) // "." // suffix
      else
         mci%grid_filename = suffix // "_m" // trim (buffer) // ".vg"
      end if
   end if
end subroutine mci_vamp_record_index
```

Set the grid parameters.
⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: set_grid_parameters => mci_vamp_set_grid_parameters
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_set_grid_parameters (mci, grid_par)
      class(mci_vamp_t), intent(inout) :: mci
      type(grid_parameters_t), intent(in) :: grid_par
      mci%grid_par = grid_par
      mci%min_calls = grid_par%min_calls_per_bin * mci%n_channel
    end subroutine mci_vamp_set_grid_parameters
```

Set the history parameters.
⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: set_history_parameters => mci_vamp_set_history_parameters
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_set_history_parameters (mci, history_par)
      class(mci_vamp_t), intent(inout) :: mci
      type(history_parameters_t), intent(in) :: history_par
      mci%history_par = history_par
    end subroutine mci_vamp_set_history_parameters
```

Set the rebuild flag, also the flag for checking the grid file.
⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: set_rebuild_flag => mci_vamp_set_rebuild_flag
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_set_rebuild_flag (mci, rebuild, check_grid_file)
      class(mci_vamp_t), intent(inout) :: mci
      logical, intent(in) :: rebuild
      logical, intent(in) :: check_grid_file
      mci%rebuild = rebuild
      mci%check_grid_file = check_grid_file
    end subroutine mci_vamp_set_rebuild_flag
```

Set the filename.
⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: set_grid_filename => mci_vamp_set_grid_filename
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_set_grid_filename (mci, name, run_id)
  class(mci_vamp_t), intent(inout) :: mci
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: run_id
  if (present (run_id)) then
     mci%grid_filename = name // "." // run_id // ".vg"
  else
     mci%grid_filename = name // ".vg"
  end if
  mci%grid_filename_set = .true.
end subroutine mci_vamp_set_grid_filename
```

Declare particular dimensions as flat.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: declare_flat_dimensions => mci_vamp_declare_flat_dimensions
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_declare_flat_dimensions (mci, dim_flat)
  class(mci_vamp_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: dim_flat
  integer :: d
  allocate (mci%dim_is_flat (mci%n_dim), source = .false.)
  do d = 1, size (dim_flat)
     mci%dim_is_flat(dim_flat(d)) = .true.
  end do
end subroutine mci_vamp_declare_flat_dimensions
```

Declare equivalences. We have an array of channel equivalences, provided by the phase-space module. Here, we translate this into the `vamp_equivalences` array.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: declare_equivalences => mci_vamp_declare_equivalences
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_declare_equivalences (mci, channel, dim_offset)
  class(mci_vamp_t), intent(inout) :: mci
  type(phs_channel_t), dimension(:), intent(in) :: channel
  integer, intent(in) :: dim_offset
  integer, dimension(:), allocatable :: perm, mode
  integer :: n_channels, n_dim, n_equivalences
  integer :: c, i, j, left, right
  n_channels = mci%n_channel
  n_dim = mci%n_dim
  n_equivalences = 0
  do c = 1, n_channels
     n_equivalences = n_equivalences + size (channel(c)%eq)
  end do
  call vamp_equivalences_init (mci%equivalences, &
       n_equivalences, n_channels, n_dim)
  allocate (perm (n_dim))
  allocate (mode (n_dim))
  perm(1:dim_offset) = [(i, i = 1, dim_offset)]
  mode(1:dim_offset) = VEQ_IDENTITY
```

```
      c = 1
      j = 0
      do i = 1, n_equivalences
         if (j < size (channel(c)%eq)) then
            j = j + 1
         else
            c = c + 1
            j = 1
         end if
         associate (eq => channel(c)%eq(j))
           left = c
           right = eq%c
           perm(dim_offset+1:) = eq%perm + dim_offset
           mode(dim_offset+1:) = eq%mode
           call vamp_equivalence_set (mci%equivalences, &
                 i, left, right, perm, mode)
         end associate
      end do
      call vamp_equivalences_complete (mci%equivalences)
    end subroutine mci_vamp_declare_equivalences
```

Allocate instance with matching type.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: allocate_instance => mci_vamp_allocate_instance
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_allocate_instance (mci, mci_instance)
      class(mci_vamp_t), intent(in) :: mci
      class(mci_instance_t), intent(out), pointer :: mci_instance
      allocate (mci_vamp_instance_t :: mci_instance)
    end subroutine mci_vamp_allocate_instance
```

Allocate a new integration pass. We can preset everything that does not depend on the number of iterations and calls. This is postponed to the `integrate` method.

In the final pass, we do not check accuracy goal etc., since we can assume that the user wants to perform and average all iterations in this pass.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: add_pass => mci_vamp_add_pass
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_add_pass (mci, adapt_grids, adapt_weights, final)
      class(mci_vamp_t), intent(inout) :: mci
      logical, intent(in), optional :: adapt_grids, adapt_weights, final
      integer :: i_pass, i_it
      type(pass_t), pointer :: new
      allocate (new)
      if (associated (mci%current_pass)) then
         i_pass = mci%current_pass%i_pass + 1
         i_it   = mci%current_pass%i_first_it + mci%current_pass%n_it
         mci%current_pass%next => new
      else
         i_pass = 1
```

```
      i_it = 1
      mci%first_pass => new
   end if
   mci%current_pass => new
   new%i_pass = i_pass
   new%i_first_it = i_it
   if (present (adapt_grids)) then
      new%adapt_grids = adapt_grids
   else
      new%adapt_grids = .false.
   end if
   if (present (adapt_weights)) then
      new%adapt_weights = adapt_weights
   else
      new%adapt_weights = .false.
   end if
   if (present (final)) then
      new%is_final_pass = final
   else
      new%is_final_pass = .false.
   end if
end subroutine mci_vamp_add_pass
```

Update the list of integration passes. All passes except for the last one must match exactly. For the last one, integration results are updated. The reference output may contain extra passes, these are ignored.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
   procedure :: update_from_ref => mci_vamp_update_from_ref
```

⟨*MCI vamp: procedures*⟩+≡
```
   subroutine mci_vamp_update_from_ref (mci, mci_ref, success)
     class(mci_vamp_t), intent(inout) :: mci
     class(mci_t), intent(in) :: mci_ref
     logical, intent(out) :: success
     type(pass_t), pointer :: current_pass, ref_pass
     select type (mci_ref)
     type is (mci_vamp_t)
        current_pass => mci%first_pass
        ref_pass => mci_ref%first_pass
        success = .true.
        do while (success .and. associated (current_pass))
           if (associated (ref_pass)) then
              if (associated (current_pass%next)) then
                 success = current_pass .matches. ref_pass
              else
                 call current_pass%update (ref_pass, success)
                 if (current_pass%integral_defined) then
                    mci%integral = current_pass%get_integral ()
                    mci%error = current_pass%get_error ()
                    mci%efficiency = current_pass%get_efficiency ()
                    mci%integral_known = .true.
                    mci%error_known = .true.
                    mci%efficiency_known = .true.
                 end if
```

```
           end if
           current_pass => current_pass%next
           ref_pass => ref_pass%next
       else
           success = .false.
       end if
   end do
   end select
end subroutine mci_vamp_update_from_ref
```

Update the MCI record (i.e., the integration passes) by reading from input stream. The stream should contain a `write` output from a previous run. We first check the MD5 sum of the configuration parameters. If that matches, we proceed directly to the stored integration passes. If successful, we may continue to read the file; the position will be after a blank line that must follow the MCI record.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
   procedure :: update => mci_vamp_update
```

⟨*MCI vamp: procedures*⟩+≡
```
   subroutine mci_vamp_update (mci, u, success)
     class(mci_vamp_t), intent(inout) :: mci
     integer, intent(in) :: u
     logical, intent(out) :: success
     character(80) :: buffer
     character(32) :: md5sum_file
     type(mci_vamp_t) :: mci_file
     integer :: n_pass, n_it
     call read_sval (u, md5sum_file)
     if (mci%check_grid_file) then
        success = md5sum_file == mci%md5sum
     else
        success = .true.
     end if
     if (success) then
        read (u, *)
        read (u, "(A)")  buffer
        if (trim (adjustl (buffer)) == "VAMP integrator:") then
           n_pass = 0
           n_it = 0
           do
              read (u, "(A)")  buffer
              select case (trim (adjustl (buffer)))
              case ("")
                 exit
              case ("Integration pass:")
                 call mci_file%add_pass ()
                 call mci_file%current_pass%read (u, n_pass, n_it)
                 n_pass = n_pass + 1
                 n_it = n_it + mci_file%current_pass%n_it
              end select
           end do
           call mci%update_from_ref (mci_file, success)
           call mci_file%final ()
```

```
            else
               call msg_fatal ("VAMP: reading grid file: corrupted data")
            end if
         end if
      end subroutine mci_vamp_update
```

Read / write grids from / to file.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
      procedure :: write_grids => mci_vamp_write_grids
      procedure :: read_grids_header => mci_vamp_read_grids_header
      procedure :: read_grids_data => mci_vamp_read_grids_data
      procedure :: read_grids => mci_vamp_read_grids
```

⟨*MCI vamp: procedures*⟩+≡

```
      subroutine mci_vamp_write_grids (mci, instance)
        class(mci_vamp_t), intent(in) :: mci
        class(mci_instance_t), intent(inout) :: instance
        integer :: u
        select type (instance)
        type is (mci_vamp_instance_t)
           if (mci%grid_filename_set) then
              if (instance%grids_defined) then
                 u = free_unit ()
                 open (u, file = char (mci%grid_filename), &
                      action = "write", status = "replace")
                 write (u, "(1x,A,A,A)")  "MD5sum = '", mci%md5sum, "'"
                 write (u, *)
                 call mci%write (u)
                 write (u, *)
                 write (u, "(1x,A)")  "VAMP grids:"
                 call vamp_write_grids (instance%grids, u, &
                      write_integrals = .true.)
                 close (u)
              else
                 call msg_bug ("VAMP: write grids: grids undefined")
              end if
           else
              call msg_bug ("VAMP: write grids: filename undefined")
           end if
        end select
      end subroutine mci_vamp_write_grids

      subroutine mci_vamp_read_grids_header (mci, success)
        class(mci_vamp_t), intent(inout) :: mci
        logical, intent(out) :: success
        logical :: exist
        integer :: u
        success = .false.
        if (mci%grid_filename_set) then
           inquire (file = char (mci%grid_filename), exist = exist)
           if (exist) then
              u = free_unit ()
              open (u, file = char (mci%grid_filename), &
                   action = "read", status = "old")
```

```
         call mci%update (u, success)
         close (u)
         if (.not. success) then
            write (msg_buffer, "(A,A,A)") &
                 "VAMP: parameter mismatch, discarding grid file '", &
                 char (mci%grid_filename), "'"
            call msg_message ()
         end if
      end if
   else
      call msg_bug ("VAMP: read grids: filename undefined")
   end if
end subroutine mci_vamp_read_grids_header

subroutine mci_vamp_read_grids_data (mci, instance, read_integrals)
  class(mci_vamp_t), intent(in) :: mci
  class(mci_instance_t), intent(inout) :: instance
  logical, intent(in), optional :: read_integrals
  integer :: u
  character(80) :: buffer
  select type (instance)
  type is (mci_vamp_instance_t)
     if (.not. instance%grids_defined) then
        u = free_unit ()
        open (u, file = char (mci%grid_filename), &
             action = "read", status = "old")
        do
           read (u, "(A)")  buffer
           if (trim (adjustl (buffer)) == "VAMP grids:")  exit
        end do
        call vamp_read_grids (instance%grids, u, read_integrals)
        close (u)
        instance%grids_defined = .true.
     else
        call msg_bug ("VAMP: read grids: grids already defined")
     end if
  end select
end subroutine mci_vamp_read_grids_data

subroutine mci_vamp_read_grids (mci, instance, success)
  class(mci_vamp_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  logical, intent(out) :: success
  logical :: exist
  integer :: u
  character(80) :: buffer
  select type (instance)
  type is (mci_vamp_instance_t)
     success = .false.
     if (mci%grid_filename_set) then
        if (.not. instance%grids_defined) then
           inquire (file = char (mci%grid_filename), exist = exist)
           if (exist) then
              u = free_unit ()
```

```
                    open (u, file = char (mci%grid_filename), &
                         action = "read", status = "old")
                    call mci%update (u, success)
                    if (success) then
                        read (u, "(A)")  buffer
                        if (trim (adjustl (buffer)) == "VAMP grids:") then
                            call vamp_read_grids (instance%grids, u)
                        else
                            call msg_fatal ("VAMP: reading grid file: &
                                   &corrupted grid data")
                        end if
                    else
                        write (msg_buffer, "(A,A,A)") &
                               "VAMP: parameter mismatch, discarding grid file '", &
                               char (mci%grid_filename), "'"
                        call msg_message ()
                    end if
                    close (u)
                    instance%grids_defined = success
                 end if
            else
                call msg_bug ("VAMP: read grids: grids already defined")
            end if
        else
            call msg_bug ("VAMP: read grids: filename undefined")
        end if
      end select
   end subroutine mci_vamp_read_grids
```

Auxiliary: Read real, integer, string value. We search for an equals sign, the value must follow.

⟨*MCI vamp: procedures*⟩+≡

```
   subroutine read_rval (u, rval)
     integer, intent(in) :: u
     real(default), intent(out) :: rval
     character(80) :: buffer
     read (u, "(A)")  buffer
     buffer = adjustl (buffer(scan (buffer, "=") + 1:))
     read (buffer, *)  rval
   end subroutine read_rval

   subroutine read_ival (u, ival)
     integer, intent(in) :: u
     integer, intent(out) :: ival
     character(80) :: buffer
     read (u, "(A)")  buffer
     buffer = adjustl (buffer(scan (buffer, "=") + 1:))
     read (buffer, *)  ival
   end subroutine read_ival

   subroutine read_sval (u, sval)
     integer, intent(in) :: u
     character(*), intent(out) :: sval
```

```
      character(80) :: buffer
      read (u, "(A)")  buffer
      buffer = adjustl (buffer(scan (buffer, "=") + 1:))
      read (buffer, *)  sval
    end subroutine read_sval

    subroutine read_lval (u, lval)
      integer, intent(in) :: u
      logical, intent(out) :: lval
      character(80) :: buffer
      read (u, "(A)")  buffer
      buffer = adjustl (buffer(scan (buffer, "=") + 1:))
      read (buffer, *)  lval
    end subroutine read_lval
```

Integrate. Perform a new integration pass (possibly reusing previous results), which may consist of several iterations.

Note: we record the integral once per iteration. The integral stored in the `mci` record itself is the last integral of the current iteration, no averaging done. The `results` record may average results.

Note: recording the efficiency is not supported yet.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: integrate => mci_vamp_integrate
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_integrate (mci, instance, sampler, &
        n_it, n_calls, results, pacify)
      class(mci_vamp_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      integer, intent(in) :: n_it
      integer, intent(in) :: n_calls
      class(mci_results_t), intent(inout), optional :: results
      logical, intent(in), optional :: pacify
      integer :: it
      logical :: reshape, from_file, success
      select type (instance)
      type is (mci_vamp_instance_t)
         if (associated (mci%current_pass)) then
            mci%current_pass%integral_defined = .false.
            call mci%current_pass%configure (n_it, n_calls, &
                mci%min_calls, mci%grid_par%min_bins, &
                mci%grid_par%max_bins, &
                mci%grid_par%min_calls_per_channel * mci%n_channel)
            call mci%current_pass%configure_history &
                (mci%n_channel, mci%history_par)
            instance%pass_complete = .false.
            instance%it_complete = .false.
            call instance%new_pass (reshape)
            if (.not. instance%grids_defined .or. instance%grids_from_file) then
               if (mci%grid_filename_set .and. .not. mci%rebuild) then
                  call mci%read_grids_header (success)
                  from_file = success
                  if (.not. instance%grids_defined .and. success) then
```

1671

```
                call mci%read_grids_data (instance)
             end if
        else
             from_file = .false.
        end if
    else
        from_file = .false.
    end if
    if (from_file) then
        if (.not. mci%check_grid_file) &
             call msg_warning ("Reading grid file: MD5 sum check disabled")
        call msg_message ("VAMP: " &
             // "using grids and results from file '" &
             // char (mci%grid_filename) // "'")
    else if (.not. instance%grids_defined) then
        call instance%create_grids ()
    end if
    do it = 1, instance%n_it
        if (signal_is_pending ())  return
        instance%grids_from_file = from_file .and. &
             it <= mci%current_pass%get_integration_index ()
        if (.not. instance%grids_from_file) then
             instance%it_complete = .false.
             call instance%adapt_grids ()
             if (signal_is_pending ())  return
             call instance%adapt_weights ()
             if (signal_is_pending ())  return
             call instance%discard_integrals (reshape)
             if (mci%grid_par%use_vamp_equivalences) then
                call instance%sample_grids (mci%rng, sampler, &
                     mci%equivalences)
             else
                call instance%sample_grids (mci%rng, sampler)
             end if
             if (signal_is_pending ())  return
             instance%it_complete = .true.
             if (instance%integral /= 0) then
                mci%current_pass%calls(it) = instance%calls
                mci%current_pass%integral(it) = instance%integral
                if (abs (instance%error / instance%integral) &
                     > epsilon (1._default)) then
                   mci%current_pass%error(it) = instance%error
                end if
                mci%current_pass%efficiency(it) = instance%efficiency
             end if
             mci%current_pass%integral_defined = .true.
        end if
        if (present (results)) then
             if (mci%has_chains ()) then
                call mci%collect_chain_weights (instance%w)
                call results%record (1, &
                     n_calls    = mci%current_pass%calls(it), &
                     integral   = mci%current_pass%integral(it), &
                     error      = mci%current_pass%error(it), &
```

```
                        efficiency = mci%current_pass%efficiency(it), &
                        chain_weights = mci%chain_weights, &
                        suppress = pacify)
                 else
                    call results%record (1, &
                          n_calls    = mci%current_pass%calls(it), &
                          integral   = mci%current_pass%integral(it), &
                          error      = mci%current_pass%error(it), &
                          efficiency = mci%current_pass%efficiency(it), &
                          suppress = pacify)
                 end if
              end if
              if (.not. instance%grids_from_file &
                   .and. mci%grid_filename_set) then
                 call mci%write_grids (instance)
              end if
              call instance%allow_adaptation ()
              reshape = .false.
              if (.not. mci%current_pass%is_final_pass) then
                 call mci%check_goals (it, success)
                 if (success)  exit
              end if
           end do
           if (signal_is_pending ())  return
           instance%pass_complete = .true.
           mci%integral = mci%current_pass%get_integral()
           mci%error = mci%current_pass%get_error()
           mci%efficiency = mci%current_pass%get_efficiency()
           mci%integral_known = .true.
           mci%error_known = .true.
           mci%efficiency_known = .true.
           call mci%compute_md5sum (pacify)
        else
           call msg_bug ("MCI integrate: current_pass object not allocated")
        end if
     end select
   end subroutine mci_vamp_integrate
```

Check whether we are already finished with this pass.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: check_goals => mci_vamp_check_goals
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_check_goals (mci, it, success)
      class(mci_vamp_t), intent(inout) :: mci
      integer, intent(in) :: it
      logical, intent(out) :: success
      success = .false.
      if (mci%error_reached (it)) then
         mci%current_pass%n_it = it
         call msg_message ("VAMP: error goal reached; &
              &skipping iterations")
         success = .true.
         return
```

```
         end if
         if (mci%rel_error_reached (it)) then
            mci%current_pass%n_it = it
            call msg_message ("VAMP: relative error goal reached; &
                 &skipping iterations")
            success = .true.
            return
         end if
         if (mci%accuracy_reached (it)) then
            mci%current_pass%n_it = it
            call msg_message ("VAMP: accuracy goal reached; &
                 &skipping iterations")
            success = .true.
            return
         end if
    end subroutine mci_vamp_check_goals
```

Return true if the error, relative error, or accuracy goal has been reached, if
any.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
    procedure :: error_reached => mci_vamp_error_reached
    procedure :: rel_error_reached => mci_vamp_rel_error_reached
    procedure :: accuracy_reached => mci_vamp_accuracy_reached
```

⟨*MCI vamp: procedures*⟩+≡

```
    function mci_vamp_error_reached (mci, it) result (flag)
      class(mci_vamp_t), intent(in) :: mci
      integer, intent(in) :: it
      logical :: flag
      real(default) :: error_goal, error
      error_goal = mci%grid_par%error_goal
      if (error_goal > 0) then
         associate (pass => mci%current_pass)
           if (pass%integral_defined) then
              error = abs (pass%error(it))
              flag = error < error_goal
           else
              flag = .false.
           end if
         end associate
      else
         flag = .false.
      end if
    end function mci_vamp_error_reached

    function mci_vamp_rel_error_reached (mci, it) result (flag)
      class(mci_vamp_t), intent(in) :: mci
      integer, intent(in) :: it
      logical :: flag
      real(default) :: rel_error_goal, rel_error
      rel_error_goal = mci%grid_par%rel_error_goal
      if (rel_error_goal > 0) then
         associate (pass => mci%current_pass)
           if (pass%integral_defined) then
```

```
              if (pass%integral(it) /= 0) then
                  rel_error = abs (pass%error(it) / pass%integral(it))
                  flag = rel_error < rel_error_goal
              else
                  flag = .true.
              end if
          else
              flag = .false.
          end if
      end associate
    else
       flag = .false.
    end if
  end function mci_vamp_rel_error_reached

  function mci_vamp_accuracy_reached (mci, it) result (flag)
    class(mci_vamp_t), intent(in) :: mci
    integer, intent(in) :: it
    logical :: flag
    real(default) :: accuracy_goal, accuracy
    accuracy_goal = mci%grid_par%accuracy_goal
    if (accuracy_goal > 0) then
        associate (pass => mci%current_pass)
          if (pass%integral_defined) then
              if (pass%integral(it) /= 0) then
                  accuracy = abs (pass%error(it) / pass%integral(it)) &
                       * sqrt (real (pass%calls(it), default))
                  flag = accuracy < accuracy_goal
              else
                  flag = .true.
              end if
          else
              flag = .false.
          end if
      end associate
    else
       flag = .false.
    end if
  end function mci_vamp_accuracy_reached
```

Prepare an event generation pass. Should be called before a sequence of events is generated, then we should call the corresponding finalizer.

The pass-specific data of the previous integration pass are retained, but we reset the number of iterations and calls to zero. The latter now counts the number of events (calls to the sampling function, actually).

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: prepare_simulation => mci_vamp_prepare_simulation
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_prepare_simulation (mci)
      class(mci_vamp_t), intent(inout) :: mci
      logical :: success
      if (mci%grid_filename_set) then
         call mci%read_grids_header (success)
```

```
      call mci%compute_md5sum ()
      if (.not. success) then
         call msg_fatal ("Simulate: " &
               // "reading integration grids from file '" &
               // char (mci%grid_filename) // "' failed")
      end if
   else
      call msg_bug ("VAMP: simulation: no grids, no grid filename")
   end if
end subroutine mci_vamp_prepare_simulation
```

Generate weighted event. Note that the event weight (`vamp_weight`) is not just the MCI weight. `vamp_next_event` selects a channel based on the channel weights multiplied by the (previously recorded) maximum integrand value of the channel. The MCI weight is renormalized accordingly, to cancel this effect on the result.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
   procedure :: generate_weighted_event => mci_vamp_generate_weighted_event
```

⟨*MCI vamp: procedures*⟩+≡
```
   subroutine mci_vamp_generate_weighted_event (mci, instance, sampler)
     class(mci_vamp_t), intent(inout) :: mci
     class(mci_instance_t), intent(inout), target :: instance
     class(mci_sampler_t), intent(inout), target :: sampler
     class(vamp_data_t), allocatable :: data
     type(exception) :: vamp_exception
     select type (instance)
     type is (mci_vamp_instance_t)
        instance%vamp_weight_set = .false.
        allocate (mci_workspace_t :: data)
        select type (data)
        type is (mci_workspace_t)
           data%sampler => sampler
           data%instance => instance
        end select
        select type (rng => mci%rng)
        type is (rng_tao_t)
           if (instance%grids_defined) then
              call vamp_next_event ( &
                   instance%vamp_x, &
                   rng%state, &
                   instance%grids, &
                   vamp_sampling_function, &
                   data, &
                   phi = phi_trivial, &
                   weight = instance%vamp_weight, &
                   exc = vamp_exception)
              call handle_vamp_exception (vamp_exception, mci%verbose)
              instance%vamp_excess = 0
              instance%vamp_weight_set = .true.
           else
              call msg_bug ("VAMP: generate event: grids undefined")
           end if
        class default
```

```
              call msg_fatal ("VAMP event generation: &
                     &random-number generator must be TAO")
           end select
        end select
     end subroutine mci_vamp_generate_weighted_event
```

Generate unweighted event.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
     procedure :: generate_unweighted_event => &
          mci_vamp_generate_unweighted_event
```

⟨*MCI vamp: procedures*⟩+≡
```
     subroutine mci_vamp_generate_unweighted_event (mci, instance, sampler)
       class(mci_vamp_t), intent(inout) :: mci
       class(mci_instance_t), intent(inout), target :: instance
       class(mci_sampler_t), intent(inout), target :: sampler
       class(vamp_data_t), allocatable :: data
       type(exception) :: vamp_exception
       select type (instance)
       type is (mci_vamp_instance_t)
          instance%vamp_weight_set = .false.
          allocate (mci_workspace_t :: data)
          select type (data)
          type is (mci_workspace_t)
             data%sampler => sampler
             data%instance => instance
          end select
          select type (rng => mci%rng)
          type is (rng_tao_t)
             if (instance%grids_defined) then
                REJECTION: do
                   call vamp_next_event ( &
                        instance%vamp_x, &
                        rng%state, &
                        instance%grids, &
                        vamp_sampling_function, &
                        data, &
                        phi = phi_trivial, &
                        excess = instance%vamp_excess, &
                        exc = vamp_exception)
                   if (signal_is_pending ())  return
                   if (sampler%is_valid ())  exit REJECTION
                end do REJECTION
                call handle_vamp_exception (vamp_exception, mci%verbose)
                instance%vamp_weight = 1
                instance%vamp_weight_set = .true.
             else
                call msg_bug ("VAMP: generate event: grids undefined")
             end if
          class default
             call msg_fatal ("VAMP event generation: &
                    &random-number generator must be TAO")
          end select
       end select
```

```
      end subroutine mci_vamp_generate_unweighted_event
```

Rebuild an event, using the `state` input.

Note: This feature is currently unused.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: rebuild_event => mci_vamp_rebuild_event
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_rebuild_event (mci, instance, sampler, state)
      class(mci_vamp_t), intent(inout) :: mci
      class(mci_instance_t), intent(inout) :: instance
      class(mci_sampler_t), intent(inout) :: sampler
      class(mci_state_t), intent(in) :: state
      call msg_bug ("MCI vamp rebuild event not implemented yet")
    end subroutine mci_vamp_rebuild_event
```

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
    procedure :: pacify => mci_vamp_pacify
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_pacify (object, efficiency_reset, error_reset)
      class(mci_vamp_t), intent(inout) :: object
      logical, intent(in), optional :: efficiency_reset, error_reset
      logical :: err_reset
      type(pass_t), pointer :: current_pass
      err_reset = .false.
      if (present (error_reset))  err_reset = error_reset
      current_pass => object%first_pass
      do while (associated (current_pass))
         if (allocated (current_pass%error) .and. err_reset) then
            current_pass%error = 0
         end if
         if (allocated (current_pass%efficiency) .and. err_reset) then
            current_pass%efficiency = 1
         end if
         current_pass => current_pass%next
      end do
    end subroutine mci_vamp_pacify
```

13.3.5   Sampler as Workspace

In the full setup, the sampling function requires the process instance object as
workspace. We implement this by (i) implementing the process instance as a
type extension of the abstract `sampler_t` object used by the MCI implemen-
tation and (ii) providing such an object as an extra argument to the sampling
function that VAMP can call. To minimize cross-package dependencies, we use
an abstract type `vamp_workspace` that VAMP declares and extend this by in-
cluding a pointer to the `sampler` and `instance` objects. In the body of the
sampling function, we dereference this pointer and can then work with the con-
tents.

⟨*MCI vamp: types*⟩+≡

```
type, extends (vamp_data_t) :: mci_workspace_t
   class(mci_sampler_t), pointer :: sampler => null ()
   class(mci_vamp_instance_t), pointer :: instance => null ()
end type mci_workspace_t
```

13.3.6   Integrator instance

The history entries should point to the corresponding history entry in the `pass_t` object. If there is none, we may allocate a local history, which is then just transient.

⟨*MCI vamp: public*⟩+≡
```
   public :: mci_vamp_instance_t
```

⟨*MCI vamp: types*⟩+≡
```
   type, extends (mci_instance_t) :: mci_vamp_instance_t
      type(mci_vamp_t), pointer :: mci => null ()
      logical :: grids_defined = .false.
      logical :: grids_from_file = .false.
      integer :: n_it = 0
      integer :: it = 0
      logical :: pass_complete = .false.
      integer :: n_calls = 0
      integer :: calls = 0
      logical :: it_complete = .false.
      logical :: enable_adapt_grids = .false.
      logical :: enable_adapt_weights = .false.
      logical :: allow_adapt_grids = .false.
      logical :: allow_adapt_weights = .false.
      logical :: negative_weights = .false.
      integer :: n_adapt_grids = 0
      integer :: n_adapt_weights = 0
      logical :: generating_events = .false.
      real(default) :: safety_factor = 1
      type(vamp_grids) :: grids
      real(default) :: g = 0
      real(default), dimension(:), allocatable :: gi
      real(default) :: integral = 0
      real(default) :: error = 0
      real(default) :: efficiency = 0
      real(default), dimension(:), allocatable :: vamp_x
      logical :: vamp_weight_set = .false.
      real(default) :: vamp_weight = 0
      real(default) :: vamp_excess = 0
      logical :: allocate_global_history = .false.
      type(vamp_history), dimension(:), pointer :: v_history => null ()
      logical :: allocate_channel_history = .false.
      type(vamp_history), dimension(:,:), pointer :: v_histories => null ()
    contains
    ⟨MCI vamp: mci vamp instance: TBP⟩
   end type mci_vamp_instance_t
```

Output.

⟨*MCI vamp: mci vamp instance: TBP*⟩≡
```
    procedure :: write => mci_vamp_instance_write
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_write (object, unit, pacify)
      class(mci_vamp_instance_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      integer :: u, i
      character(len=7) :: fmt
      call pac_fmt (fmt, FMT_17, FMT_14, pacify)
      u = output_unit (unit)
      write (u, "(3x,A," // FMT_19 // ")") "Integrand = ", object%integrand
      write (u, "(3x,A," // FMT_19 // ")") "Weight    = ", object%mci_weight
      if (object%vamp_weight_set) then
         write (u, "(3x,A," // FMT_19 // ")") "VAMP wgt  = ", object%vamp_weight
         if (object%vamp_excess /= 0) then
            write (u, "(3x,A," // FMT_19 // ")") "VAMP exc  = ", &
                 object%vamp_excess
         end if
      end if
      write (u, "(3x,A,L1)")  "adapt grids   = ", object%enable_adapt_grids
      write (u, "(3x,A,L1)")  "adapt weights = ", object%enable_adapt_weights
      if (object%grids_defined) then
         if (object%grids_from_file) then
            write (u, "(3x,A)")  "VAMP grids: read from file"
         else
            write (u, "(3x,A)")  "VAMP grids: defined"
         end if
      else
         write (u, "(3x,A)")  "VAMP grids: [undefined]"
      end if
      write (u, "(3x,A,I0)")  "n_it          = ", object%n_it
      write (u, "(3x,A,I0)")  "it            = ", object%it
      write (u, "(3x,A,L1)")  "pass complete = ", object%it_complete
      write (u, "(3x,A,I0)")  "n_calls       = ", object%n_calls
      write (u, "(3x,A,I0)")  "calls         = ", object%calls
      write (u, "(3x,A,L1)")  "it complete   = ", object%it_complete
      write (u, "(3x,A,I0)")  "n adapt.(g)   = ", object%n_adapt_grids
      write (u, "(3x,A,I0)")  "n adapt.(w)   = ", object%n_adapt_weights
      write (u, "(3x,A,L1)")  "gen. events   = ", object%generating_events
      write (u, "(3x,A,L1)")  "neg. weights  = ", object%negative_weights
      if (object%safety_factor /= 1)  write &
            (u, "(3x,A," // fmt // ")")  "safety f = ", object%safety_factor
      write (u, "(3x,A," // fmt // ")")  "integral = ", object%integral
      write (u, "(3x,A," // fmt // ")")  "error    = ", object%error
      write (u, "(3x,A," // fmt // ")")  "eff.     = ", object%efficiency
      write (u, "(3x,A)")  "weights:"
      do i = 1, size (object%w)
         write (u, "(5x,I0,1x," // FMT_12 // ")")  i, object%w(i)
      end do
    end subroutine mci_vamp_instance_write
```

Write the grids to the specified unit.

⟨MCI vamp: mci vamp instance: TBP⟩+≡
```
    procedure :: write_grids => mci_vamp_instance_write_grids
```

⟨MCI vamp: procedures⟩+≡
```
    subroutine mci_vamp_instance_write_grids (object, unit)
      class(mci_vamp_instance_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (object%grids_defined) then
          call vamp_write_grids (object%grids, u, write_integrals = .true.)
      end if
    end subroutine mci_vamp_instance_write_grids
```

Finalizer: the history arrays are pointer arrays and need finalization.

⟨MCI vamp: mci vamp instance: TBP⟩+≡
```
    procedure :: final => mci_vamp_instance_final
```

⟨MCI vamp: tests⟩≡
```
    subroutine mci_vamp_instance_final (object)
      class(mci_vamp_instance_t), intent(inout) :: object
      if (object%allocate_global_history) then
          if (associated (object%v_history)) then
              call vamp_delete_history (object%v_history)
              deallocate (object%v_history)
          end if
      end if
      if (object%allocate_channel_history) then
          if (associated (object%v_histories)) then
              call vamp_delete_history (object%v_histories)
              deallocate (object%v_histories)
          end if
      end if
      if (object%grids_defined) then
          call vamp_delete_grids (object%grids)
          object%grids_defined = .false.
      end if
    end subroutine mci_vamp_instance_final
```

Initializer.

⟨MCI vamp: mci vamp instance: TBP⟩+≡
```
    procedure :: init => mci_vamp_instance_init
```

⟨MCI vamp: procedures⟩+≡
```
    subroutine mci_vamp_instance_init (mci_instance, mci)
      class(mci_vamp_instance_t), intent(out) :: mci_instance
      class(mci_t), intent(in), target :: mci
      call mci_instance%base_init (mci)
      select type (mci)
      type is (mci_vamp_t)
          mci_instance%mci => mci
          allocate (mci_instance%gi (mci%n_channel))
          mci_instance%allocate_global_history = .not. mci%history_par%global
```

```
         mci_instance%allocate_channel_history = .not. mci%history_par%channel
         mci_instance%negative_weights = mci%negative_weights
      end select
   end subroutine mci_vamp_instance_init
```

Prepare a new integration pass: write the pass-specific settings to the `instance` object. This should be called initially, together with the `create_grids` procedure, and whenever we start a new integration pass.

Set `reshape` if the number of calls is different than previously (unless it was zero, indicating the first pass).

We link VAMP histories to the allocated histories in the current pass object, so the recorded results are persistent. However, if there are no histories present there, we allocate them locally. In that case, the histories will disappear together with the MCI instance object.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
      procedure :: new_pass => mci_vamp_instance_new_pass
```

⟨*MCI vamp: procedures*⟩+≡
```
      subroutine mci_vamp_instance_new_pass (instance, reshape)
        class(mci_vamp_instance_t), intent(inout) :: instance
        logical, intent(out) :: reshape
        type(pass_t), pointer :: current
        associate (mci => instance%mci)
          current => mci%current_pass
          instance%n_it = current%n_it
          if (instance%n_calls == 0) then
             reshape = .false.
             instance%n_calls = current%n_calls
          else if (instance%n_calls == current%n_calls) then
             reshape = .false.
          else
             reshape = .true.
             instance%n_calls = current%n_calls
          end if
          instance%it = 0
          instance%calls = 0
          instance%enable_adapt_grids = current%adapt_grids
          instance%enable_adapt_weights = current%adapt_weights
          instance%generating_events = .false.
          if (instance%allocate_global_history) then
             if (associated (instance%v_history)) then
                call vamp_delete_history (instance%v_history)
                deallocate (instance%v_history)
             end if
             allocate (instance%v_history (instance%n_it))
             call vamp_create_history (instance%v_history, verbose = .false.)
          else
             instance%v_history => current%v_history
          end if
          if (instance%allocate_channel_history) then
             if (associated (instance%v_histories)) then
                call vamp_delete_history (instance%v_histories)
                deallocate (instance%v_histories)
```

1682

```
                end if
                allocate (instance%v_histories (instance%n_it, mci%n_channel))
                call vamp_create_history (instance%v_histories, verbose = .false.)
            else
                instance%v_histories => current%v_histories
            end if
        end associate
    end subroutine mci_vamp_instance_new_pass
```

Create a grid set within the `instance` object, using the data of the current integration pass. Also reset counters that track this grid set.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: create_grids => mci_vamp_instance_create_grids
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_create_grids (instance)
      class(mci_vamp_instance_t), intent(inout) :: instance
      type (pass_t), pointer :: current
      integer, dimension(:), allocatable :: num_div
      real(default), dimension(:,:), allocatable :: region
      associate (mci => instance%mci)
        current => mci%current_pass
        allocate (num_div (mci%n_dim))
        allocate (region (2, mci%n_dim))
        region(1,:) = 0
        region(2,:) = 1
        num_div = current%n_bins
        instance%n_adapt_grids = 0
        instance%n_adapt_weights = 0
        if (.not. instance%grids_defined) then
            call vamp_create_grids (instance%grids, &
                 region, &
                 current%n_calls, &
                 weights = instance%w, &
                 num_div = num_div, &
                 stratified = mci%grid_par%stratified)
            instance%grids_defined = .true.
        else
            call msg_bug ("VAMP: create grids: grids already defined")
        end if
      end associate
    end subroutine mci_vamp_instance_create_grids
```

Reset a grid set, so we can start a fresh integration pass. In effect, we delete results of previous integrations, but keep the grid shapes, weights, and variance arrays, so adaptation is still possible. The grids are prepared for a specific number of calls (per iteration) and sampling mode (stratified/importance).

The `vamp_discard_integrals` implementation will reshape the grids only if the argument `num_calls` is present.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: discard_integrals => mci_vamp_instance_discard_integrals
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_discard_integrals (instance, reshape)
      class(mci_vamp_instance_t), intent(inout) :: instance
      logical, intent(in) :: reshape
      instance%calls = 0
      instance%integral = 0
      instance%error = 0
      instance%efficiency = 0
      associate (mci => instance%mci)
        if (instance%grids_defined) then
           if (mci%grid_par%use_vamp_equivalences) then
              if (reshape) then
                 call vamp_discard_integrals (instance%grids, &
                      num_calls = instance%n_calls, &
                      stratified = mci%grid_par%stratified, &
                      eq = mci%equivalences)
              else
                 call vamp_discard_integrals (instance%grids, &
                      stratified = mci%grid_par%stratified, &
                      eq = mci%equivalences)
              end if
           else
              if (reshape) then
                 call vamp_discard_integrals (instance%grids, &
                      num_calls = instance%n_calls, &
                      stratified = mci%grid_par%stratified)
              else
                 call vamp_discard_integrals (instance%grids, &
                      stratified = mci%grid_par%stratified)
              end if
           end if
        else
           call msg_bug ("VAMP: discard integrals: grids undefined")
        end if
      end associate
    end subroutine mci_vamp_instance_discard_integrals
```

After grids are created (with equidistant binning and equal weight), adaptation
is redundant. Therefore, we should allow it only after a complete integration
step has been performed, calling this.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: allow_adaptation => mci_vamp_instance_allow_adaptation
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_allow_adaptation (instance)
      class(mci_vamp_instance_t), intent(inout) :: instance
      instance%allow_adapt_grids = .true.
      instance%allow_adapt_weights = .true.
    end subroutine mci_vamp_instance_allow_adaptation
```

Adapt grids.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: adapt_grids => mci_vamp_instance_adapt_grids
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_adapt_grids (instance)
      class(mci_vamp_instance_t), intent(inout) :: instance
      if (instance%enable_adapt_grids .and. instance%allow_adapt_grids) then
         if (instance%grids_defined) then
            call vamp_refine_grids (instance%grids)
            instance%n_adapt_grids = instance%n_adapt_grids + 1
         else
            call msg_bug ("VAMP: adapt grids: grids undefined")
         end if
      end if
    end subroutine mci_vamp_instance_adapt_grids
```

Adapt weights. Use the variance array returned by `VAMP` for recalculating the weight array. The parameter `channel_weights_power` dampens fluctuations.

If the number of calls in a given channel falls below a user-defined threshold, the weight is not lowered further but kept at this threshold. The other channel weights are reduced accordingly.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: adapt_weights => mci_vamp_instance_adapt_weights
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_adapt_weights (instance)
      class(mci_vamp_instance_t), intent(inout) :: instance
      real(default) :: w_sum, w_avg_ch, sum_w_underflow, w_min
      integer :: n_ch, ch, n_underflow
      logical, dimension(:), allocatable :: mask, underflow
      type(exception) :: vamp_exception
      if (instance%enable_adapt_weights .and. instance%allow_adapt_weights) then
         associate (mci => instance%mci)
           if (instance%grids_defined) then
              instance%w = instance%grids%weights &
                   * vamp_get_variance (instance%grids%grids) &
                   ** mci%grid_par%channel_weights_power
              w_sum = sum (instance%w)
              if (w_sum /= 0) then
                 instance%w = instance%w / w_sum
                 if (mci%n_chain /= 0) then
                    allocate (mask (mci%n_channel))
                    do ch = 1, mci%n_chain
                       mask = mci%chain == ch
                       n_ch = count (mask)
                       if (n_ch /= 0) then
                          w_avg_ch = sum (instance%w, mask) / n_ch
                          where (mask)  instance%w = w_avg_ch
                       end if
                    end do
                 end if
                 if (mci%grid_par%threshold_calls /= 0) then
                    w_min = &
                         real (mci%grid_par%threshold_calls, default) &
                         / instance%n_calls
                    allocate (underflow (mci%n_channel))
                    underflow = instance%w /= 0 .and. abs (instance%w) < w_min
```

```
                    n_underflow = count (underflow)
                    sum_w_underflow = sum (instance%w, mask=underflow)
                    if (sum_w_underflow /= 1) then
                       where (underflow)
                          instance%w = w_min
                       elsewhere
                          instance%w = instance%w &
                               * (1 - n_underflow * w_min) / (1 - sum_w_underflow)
                       end where
                    end if
                 end if
              end if
              call vamp_update_weights (instance%grids, instance%w, &
                   exc = vamp_exception)
              call handle_vamp_exception (vamp_exception, mci%verbose)
           else
              call msg_bug ("VAMP: adapt weights: grids undefined")
           end if
        end associate
        instance%n_adapt_weights = instance%n_adapt_weights + 1
     end if
   end subroutine mci_vamp_instance_adapt_weights
```

Integration: sample the VAMP grids. The number of calls etc. are already stored inside the grids. We provide the random-number generator, the sampling function, and a link to the workspace object, which happens to contain a pointer to the sampler object. The sampler object thus becomes the workspace of the sampling function.

Note: in the current implementation, the random-number generator must be the TAO generator. This explicit dependence should be removed from the VAMP implementation.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
   procedure :: sample_grids => mci_vamp_instance_sample_grids
```

⟨*MCI vamp: procedures*⟩+≡
```
   subroutine mci_vamp_instance_sample_grids (instance, rng, sampler, eq)
     class(mci_vamp_instance_t), intent(inout), target :: instance
     class(rng_t), intent(inout) :: rng
     class(mci_sampler_t), intent(inout), target :: sampler
     type(vamp_equivalences_t), intent(in), optional :: eq
     class(vamp_data_t), allocatable :: data
     type(exception) :: vamp_exception
     allocate (mci_workspace_t :: data)
     select type (data)
     type is (mci_workspace_t)
        data%sampler => sampler
        data%instance => instance
     end select
     select type (rng)
     type is (rng_tao_t)
        instance%it = instance%it + 1
        instance%calls = 0
        if (instance%grids_defined) then
```

1686

```
                 call vamp_sample_grids ( &
                     rng%state, &
                     instance%grids, &
                     vamp_sampling_function, &
                     data, &
                     1, &
                     eq = eq, &
                     history = instance%v_history(instance%it:), &
                     histories = instance%v_histories(instance%it:,:), &
                     integral = instance%integral, &
                     std_dev = instance%error, &
                     exc = vamp_exception, &
                     negative_weights = instance%negative_weights)
                 call handle_vamp_exception (vamp_exception, instance%mci%verbose)
                 instance%efficiency = instance%get_efficiency ()
             else
                 call msg_bug ("VAMP: sample grids: grids undefined")
             end if
         class default
             call msg_fatal ("VAMP integration: random-number generator must be TAO")
         end select
     end subroutine mci_vamp_instance_sample_grids
```

Compute the reweighting efficiency for the current grids, suitable averaged over
all active channels.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
     procedure :: get_efficiency_array => mci_vamp_instance_get_efficiency_array
     procedure :: get_efficiency => mci_vamp_instance_get_efficiency
```

⟨*MCI vamp: procedures*⟩+≡
```
     function mci_vamp_instance_get_efficiency_array (mci) result (efficiency)
       class(mci_vamp_instance_t), intent(in) :: mci
       real(default), dimension(:), allocatable :: efficiency
       allocate (efficiency (mci%mci%n_channel))
       where (mci%grids%grids%f_max /= 0)
          efficiency = mci%grids%grids%mu(1) / abs (mci%grids%grids%f_max)
       elsewhere
          efficiency = 0
       end where
     end function mci_vamp_instance_get_efficiency_array

     function mci_vamp_instance_get_efficiency (mci) result (efficiency)
       class(mci_vamp_instance_t), intent(in) :: mci
       real(default) :: efficiency
       real(default), dimension(:), allocatable :: weight
       real(default) :: norm
       allocate (weight (mci%mci%n_channel))
       weight = mci%grids%weights * abs (mci%grids%grids%f_max)
       norm = sum (weight)
       if (norm /= 0) then
          efficiency = dot_product (mci%get_efficiency_array (), weight) / norm
       else
          efficiency = 1
       end if
```

```
          end function mci_vamp_instance_get_efficiency
```

Prepare an event generation pass. Should be called before a sequence of events
is generated, then we should call the corresponding finalizer.

The pass-specific data of the previous integration pass are retained, but we
reset the number of iterations and calls to zero. The latter now counts the
number of events (calls to the sampling function, actually).

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: init_simulation => mci_vamp_instance_init_simulation
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_init_simulation (instance, safety_factor)
      class(mci_vamp_instance_t), intent(inout) :: instance
      real(default), intent(in), optional :: safety_factor
      associate (mci => instance%mci)
        allocate (instance%vamp_x (mci%n_dim))
        instance%it = 0
        instance%calls = 0
        instance%generating_events = .true.
        if (present (safety_factor))  instance%safety_factor = safety_factor
        if (.not. instance%grids_defined) then
           if (mci%grid_filename_set) then
              if (.not. mci%check_grid_file) &
                   call msg_warning ("Reading grid file: MD5 sum check disabled")
              call msg_message ("Simulate: " &
                   // "using integration grids from file '" &
                   // char (mci%grid_filename) // "'")
              call mci%read_grids_data (instance)
              if (instance%safety_factor /= 1) then
                 write (msg_buffer, "(A,ES10.3,A)")  "Simulate: &
                      &applying safety factor", instance%safety_factor, &
                      " to event rejection"
                 call msg_message ()
                 instance%grids%grids%f_max = &
                      instance%grids%grids%f_max * instance%safety_factor
              end if
           else
              call msg_bug ("VAMP: simulation: no grids, no grid filename")
           end if
        end if
      end associate
    end subroutine mci_vamp_instance_init_simulation
```

Finalize an event generation pass. Should be called before a sequence of events
is generated, then we should call the corresponding finalizer.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: final_simulation => mci_vamp_instance_final_simulation
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_final_simulation (instance)
      class(mci_vamp_instance_t), intent(inout) :: instance
      if (allocated (instance%vamp_x))  deallocate (instance%vamp_x)
    end subroutine mci_vamp_instance_final_simulation
```

13.3.7 Sampling function

The VAMP sampling function has a well-defined interface which we have to implement. The `data` argument allows us to pass pointers to the `sampler` and `instance` objects, so we can access configuration data and fill point-dependent contents within these objects.

The `weights` and `channel` argument must be present in the call.

Note: we would normally declare the `instance` pointer with the concrete type, or just use the `data` component directly. Unfortunately, gfortran 4.6 forgets the inherited base-type methods in that case.

Note: this is the place where we must look for external signals, i.e., interrupt from the OS. We would like to raise a VAMP exception which is then caught by `vamp_sample_grids` as the caller, so it dumps its current state and returns (with the signal still pending). WHIZARD will then terminate gracefully. Of course, VAMP should be able to resume from the dump.

In the current implementation, we handle the exception in place and terminate immediately. The incomplete current integration pass is lost.

⟨*MCI vamp: procedures*⟩+≡

```
    function vamp_sampling_function &
        (xi, data, weights, channel, grids) result (f)
      real(default) :: f
      real(default), dimension(:), intent(in) :: xi
      class(vamp_data_t), intent(in) :: data
      real(default), dimension(:), intent(in), optional :: weights
      integer, intent(in), optional :: channel
      type(vamp_grid), dimension(:), intent(in), optional :: grids
      type(exception) :: exc
      class(mci_instance_t), pointer :: instance
      logical :: verbose
      character(*), parameter :: FN = "WHIZARD sampling function"
      select type (data)
      type is (mci_workspace_t)
         instance => data%instance
         select type (instance)
         class is (mci_vamp_instance_t)
            instance%calls = instance%calls + 1
            verbose = instance%mci%verbose
         end select
         call instance%evaluate (data%sampler, channel, xi)
         if (signal_is_pending ()) then
            call raise_exception (exc, EXC_FATAL, FN, "signal received")
            call handle_vamp_exception (exc, verbose)
            call terminate_now_if_signal ()
         end if
         f = instance%get_value ()
      end select
    end function vamp_sampling_function
```

This is supposed to be the mapping between integration channels. The VAMP event generating procedures technically require it, but it is meaningless in our setup where all transformations happen inside the sampler object. So, this implementation is trivial:

⟨*MCI vamp: procedures*⟩+≡
```
pure function phi_trivial (xi, channel_dummy) result (x)
  real(default), dimension(:), intent(in) :: xi
  integer, intent(in) :: channel_dummy
  real(default), dimension(size(xi)) :: x
  x = xi
end function phi_trivial
```

13.3.8   Integrator instance: evaluation

Here, we compute the multi-channel reweighting factor for the current channel, that accounts for the Jacobians of the transformations from/to all other channels.

The computation of the VAMP probabilities may consume considerable time, therefore we enable parallel evaluation. (Collecting the contributions to `mci%g` is a reduction, which we should also implement via OpenMP.)

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
procedure :: compute_weight => mci_vamp_instance_compute_weight
```

⟨*MCI vamp: procedures*⟩+≡
```
subroutine mci_vamp_instance_compute_weight (mci, c)
  class(mci_vamp_instance_t), intent(inout) :: mci
  integer, intent(in) :: c
  integer :: i
  mci%selected_channel = c
  !$OMP PARALLEL PRIVATE(i) SHARED(mci)
  !$OMP DO
  do i = 1, mci%mci%n_channel
     if (mci%w(i) /= 0) then
        mci%gi(i) = vamp_probability (mci%grids%grids(i), mci%x(:,i))
     else
        mci%gi(i) = 0
     end if
  end do
  !$OMP END DO
  !$OMP END PARALLEL
  mci%g = 0
  if (mci%gi(c) /= 0) then
     do i = 1, mci%mci%n_channel
        if (mci%w(i) /= 0 .and. mci%f(i) /= 0) then
           mci%g = mci%g + mci%w(i) * mci%gi(i) / mci%f(i)
        end if
     end do
  end if
  if (mci%g /= 0) then
     mci%mci_weight = mci%gi(c) / mci%g
  else
     mci%mci_weight = 0
  end if
end subroutine mci_vamp_instance_compute_weight
```

1690

Record the integrand.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: record_integrand => mci_vamp_instance_record_integrand
```

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine mci_vamp_instance_record_integrand (mci, integrand)
      class(mci_vamp_instance_t), intent(inout) :: mci
      real(default), intent(in) :: integrand
      mci%integrand = integrand
    end subroutine mci_vamp_instance_record_integrand
```

Get the event weight. The default routine returns the same value that we would use for integration. This is correct if we select the integration channel according to the channel weight. `vamp_next_event` does differently, so we should rather rely on the weight that VAMP returns. This is the value stored in `vamp_weight`. We override the default TBP accordingly.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡
```
    procedure :: get_event_weight => mci_vamp_instance_get_event_weight
    procedure :: get_event_excess => mci_vamp_instance_get_event_excess
```

⟨*MCI vamp: procedures*⟩+≡
```
    function mci_vamp_instance_get_event_weight (mci) result (value)
      class(mci_vamp_instance_t), intent(in) :: mci
      real(default) :: value
      if (mci%vamp_weight_set) then
         value = mci%vamp_weight
      else
         call msg_bug ("VAMP: attempt to read undefined event weight")
      end if
    end function mci_vamp_instance_get_event_weight

    function mci_vamp_instance_get_event_excess (mci) result (value)
      class(mci_vamp_instance_t), intent(in) :: mci
      real(default) :: value
      if (mci%vamp_weight_set) then
         value = mci%vamp_excess
      else
         call msg_bug ("VAMP: attempt to read undefined event excess weight")
      end if
    end function mci_vamp_instance_get_event_excess
```

13.3.9   VAMP exceptions

A VAMP routine may have raised an exception. Turn this into a WHIZARD error message.

An external signal could raise a fatal exception, but this should be delayed and handled by the correct termination routine.

⟨*MCI vamp: procedures*⟩+≡
```
    subroutine handle_vamp_exception (exc, verbose)
      type(exception), intent(in) :: exc
      logical, intent(in) :: verbose
      integer :: exc_level
```

```
            if (verbose) then
               exc_level = EXC_INFO
            else
               exc_level = EXC_ERROR
            end if
            if (exc%level >= exc_level) then
               write (msg_buffer, "(A,':',1x,A)")  trim (exc%origin), trim (exc%message)
               select case (exc%level)
               case (EXC_INFO);  call msg_message ()
               case (EXC_WARN);  call msg_warning ()
               case (EXC_ERROR); call msg_error ()
               case (EXC_FATAL)
                  if (signal_is_pending ()) then
                     call msg_message ()
                  else
                     call msg_fatal ()
                  end if
               end select
            end if
         end subroutine handle_vamp_exception
```

13.3.10 Unit tests

⟨*MCI vamp: public*⟩+≡
```
    public :: mci_vamp_test
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*MCI vamp: execute tests*⟩
```
    end subroutine mci_vamp_test
```

Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

In mode 1, the function is $f(x) = 3x^2$ with integral $\int_0^1 f(x)\,dx = 1$ and maximum $f(1) = 3$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

In mode 2, the function is $11x^{10}$, also with integral 1.

⟨*MCI vamp: test types*⟩≡
```
    type, extends (mci_sampler_t) :: test_sampler_1_t
       real(default), dimension(:), allocatable :: x
       real(default) :: val
       integer :: mode = 1
     contains
```
⟨*MCI vamp: test sampler 1: TBP*⟩
```
    end type test_sampler_1_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI vamp: test sampler 1: TBP*⟩≡
```
procedure :: write => test_sampler_1_write
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_1_write (object, unit, testflag)
  class(test_sampler_1_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  select case (object%mode)
  case (1)
     write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
  case (2)
     write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10"
  case (3)
     write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10 * 2 * cos^2 (2 pi y)"
  end select
end subroutine test_sampler_1_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp: test sampler 1: TBP*⟩+≡
```
procedure :: evaluate => test_sampler_1_evaluate
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
  class(test_sampler_1_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (allocated (sampler%x))  deallocate (sampler%x)
  allocate (sampler%x (size (x_in)))
  sampler%x = x_in
  select case (sampler%mode)
  case (1)
     sampler%val = 3 * x_in(1) ** 2
  case (2)
     sampler%val = 11 * x_in(1) ** 10
  case (3)
     sampler%val = 11 * x_in(1) ** 10 * 2 * cos (twopi * x_in(2)) ** 2
  end select
  call sampler%fetch (val, x, f)
end subroutine test_sampler_1_evaluate
```

The point is always valid.

⟨*MCI vamp: test sampler 1: TBP*⟩+≡
```
procedure :: is_valid => test_sampler_1_is_valid
```

⟨*MCI vamp: tests*⟩+≡
```
    function test_sampler_1_is_valid (sampler) result (valid)
      class(test_sampler_1_t), intent(in) :: sampler
      logical :: valid
      valid = .true.
    end function test_sampler_1_is_valid
```

Rebuild: compute all but the function value.
⟨*MCI vamp: test sampler 1: TBP*⟩+≡
```
    procedure :: rebuild => test_sampler_1_rebuild
```
⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
      class(test_sampler_1_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      if (allocated (sampler%x))  deallocate (sampler%x)
      allocate (sampler%x (size (x_in)))
      sampler%x = x_in
      sampler%val = val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_1_rebuild
```

Extract the results.
⟨*MCI vamp: test sampler 1: TBP*⟩+≡
```
    procedure :: fetch => test_sampler_1_fetch
```
⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_1_fetch (sampler, val, x, f)
      class(test_sampler_1_t), intent(in) :: sampler
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      val = sampler%val
      x(:,1) = sampler%x
      f = 1
    end subroutine test_sampler_1_fetch
```

Two-channel, two dimension test sampler

This sampler implements the function

$$f(x, y) = 4 \sin^2(\pi x) \sin^2(\pi y) + 2 \sin^2(\pi v) \qquad (13.11)$$

where

$$x = u^v \qquad\qquad u = xy \qquad\qquad (13.12)$$

$$y = u^{(1-v)} \qquad\qquad v = \frac{1}{2} \left(1 + \frac{\log(x/y)}{\log xy} \right) \qquad (13.13)$$

1694

Each term contributes 1 to the integral. The first term in the function is peaked along a cross aligned to the coordinates x and y, while the second term is peaked along the diagonal $x = y$.

The Jacobian is

$$\frac{\partial(x, y)}{\partial(u, v)} = |\log u| \tag{13.14}$$

⟨*MCI vamp: test types*⟩+≡
```
type, extends (mci_sampler_t) :: test_sampler_2_t
   real(default), dimension(:,:), allocatable :: x
   real(default), dimension(:), allocatable :: f
   real(default) :: val
 contains
 ⟨MCI vamp: test sampler 2: TBP⟩
end type test_sampler_2_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI vamp: test sampler 2: TBP*⟩≡
```
procedure :: write => test_sampler_2_write
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_2_write (object, unit, testflag)
  class(test_sampler_2_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)") "Two-channel test sampler 2"
end subroutine test_sampler_2_write
```

Kinematics: compute x and Jacobians, given the input parameter array.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡
```
procedure :: compute => test_sampler_2_compute
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_2_compute (sampler, c, x_in)
  class(test_sampler_2_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default) :: xx, yy, uu, vv
  if (.not. allocated (sampler%x)) &
       allocate (sampler%x (size (x_in), 2))
  if (.not. allocated (sampler%f)) &
       allocate (sampler%f (2))
  select case (c)
  case (1)
     xx = x_in(1)
     yy = x_in(2)
     uu = xx * yy
     vv = (1 + log (xx/yy) / log (xx*yy)) / 2
  case (2)
     uu = x_in(1)
     vv = x_in(2)
     xx = uu ** vv
```

```
       yy = uu ** (1 - vv)
    end select
    sampler%val = (2 * sin (pi * xx) * sin (pi * yy)) ** 2 &
         + 2 * sin (pi * vv) ** 2
    sampler%f(1) = 1
    sampler%f(2) = abs (log (uu))
    sampler%x(:,1) = [xx, yy]
    sampler%x(:,2) = [uu, vv]
 end subroutine test_sampler_2_compute
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡
```
    procedure :: evaluate => test_sampler_2_evaluate
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
      class(test_sampler_2_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      call sampler%compute (c, x_in)
      call sampler%fetch (val, x, f)
    end subroutine test_sampler_2_evaluate
```

The point is always valid.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡
```
    procedure :: is_valid => test_sampler_2_is_valid
```

⟨*MCI vamp: tests*⟩+≡
```
    function test_sampler_2_is_valid (sampler) result (valid)
      class(test_sampler_2_t), intent(in) :: sampler
      logical :: valid
      valid = .true.
    end function test_sampler_2_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡
```
    procedure :: rebuild => test_sampler_2_rebuild
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
      class(test_sampler_2_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      call sampler%compute (c, x_in)
      x = sampler%x
      f = sampler%f
```

```
      end subroutine test_sampler_2_rebuild
```

Extract the results.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡
```
    procedure :: fetch => test_sampler_2_fetch
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_2_fetch (sampler, val, x, f)
      class(test_sampler_2_t), intent(in) :: sampler
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      val = sampler%val
      x = sampler%x
      f = sampler%f
    end subroutine test_sampler_2_fetch
```

Two-channel, one dimension test sampler

This sampler implements the function

$$f(x, y) = a * 5x^4 + b * 5(1 - x)^4 \qquad (13.15)$$

Each term contributes 1 to the integral, multiplied by a or b, respectively. The first term is peaked at $x = 1$, the second one at $x = 0$..

We implement the two mappings

$$x = u^{1/5} \quad \text{and} \quad x = 1 - v^{1/5}, \qquad (13.16)$$

with Jacobians

$$\frac{\partial(x)}{\partial(u)} = u^{-4/5}/5 \quad \text{and} \quad v^{-4/5}/5, \qquad (13.17)$$

respectively. The first mapping concentrates points near $x = 1$, the second one near $x = 0$.

⟨*MCI vamp: test types*⟩+≡
```
    type, extends (mci_sampler_t) :: test_sampler_3_t
      real(default), dimension(:,:), allocatable :: x
      real(default), dimension(:), allocatable :: f
      real(default) :: val
      real(default) :: a = 1
      real(default) :: b = 1
     contains
     ⟨MCI vamp: test sampler 3: TBP⟩
    end type test_sampler_3_t
```

Output: display a and b

⟨*MCI vamp: test sampler 3: TBP*⟩≡
```
    procedure :: write => test_sampler_3_write
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_3_write (object, unit, testflag)
      class(test_sampler_3_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)") "Two-channel test sampler 3"
      write (u, "(3x,A,F5.2)")  "a = ", object%a
      write (u, "(3x,A,F5.2)")  "b = ", object%b
    end subroutine test_sampler_3_write
```

Kinematics: compute x and Jacobians, given the input parameter array.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
    procedure :: compute => test_sampler_3_compute
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_3_compute (sampler, c, x_in)
      class(test_sampler_3_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default) :: u, v, xx
      if (.not. allocated (sampler%x)) &
           allocate (sampler%x (size (x_in), 2))
      if (.not. allocated (sampler%f)) &
           allocate (sampler%f (2))
      select case (c)
      case (1)
         u = x_in(1)
         xx = u ** 0.2_default
         v = (1 - xx) ** 5._default
      case (2)
         v = x_in(1)
         xx = 1 - v ** 0.2_default
         u = xx ** 5._default
      end select
      sampler%val = sampler%a * 5 * xx ** 4 + sampler%b * 5 * (1 - xx) ** 4
      sampler%f(1) = 0.2_default * u ** (-0.8_default)
      sampler%f(2) = 0.2_default * v ** (-0.8_default)
      sampler%x(:,1) = [u]
      sampler%x(:,2) = [v]
    end subroutine test_sampler_3_compute
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
    procedure :: evaluate => test_sampler_3_evaluate
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_3_evaluate (sampler, c, x_in, val, x, f)
      class(test_sampler_3_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(out) :: val
```

1698

```
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      call sampler%compute (c, x_in)
      call sampler%fetch (val, x, f)
    end subroutine test_sampler_3_evaluate
```

The point is always valid.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
    procedure :: is_valid => test_sampler_3_is_valid
```

⟨*MCI vamp: tests*⟩+≡
```
    function test_sampler_3_is_valid (sampler) result (valid)
      class(test_sampler_3_t), intent(in) :: sampler
      logical :: valid
      valid = .true.
    end function test_sampler_3_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
    procedure :: rebuild => test_sampler_3_rebuild
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_3_rebuild (sampler, c, x_in, val, x, f)
      class(test_sampler_3_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      call sampler%compute (c, x_in)
      x = sampler%x
      f = sampler%f
    end subroutine test_sampler_3_rebuild
```

Extract the results.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
    procedure :: fetch => test_sampler_3_fetch
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine test_sampler_3_fetch (sampler, val, x, f)
      class(test_sampler_3_t), intent(in) :: sampler
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      val = sampler%val
      x = sampler%x
      f = sampler%f
    end subroutine test_sampler_3_fetch
```

One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

Note: We would like to check the precise contents of the grid allocated during integration, but the output format for reals is very long (for good reasons), so the last digits in the grid content display are numerical noise. So, we just check the integration results.

⟨*MCI vamp: execute tests*⟩≡
```
call test (mci_vamp_1, "mci_vamp_1", &
    "one-dimensional integral", &
    u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine mci_vamp_1 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_1"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension &
      &(single channel)"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 1)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_1_t :: sampler)
  call sampler%write (u)

  write (u, "(A)")
```

```
      write (u, "(A)")   "* Integrate with n_calls = 1000"
      write (u, "(A)")   "   (lower precision to avoid"
      write (u, "(A)")   "      numerical noise)"
      write (u, "(A)")

      select type (mci)
      type is (mci_vamp_t)
         call mci%add_pass ()
      end select
      call mci%integrate (mci_instance, sampler, 1, 1000, pacify = .true.)
      call mci%write (u, .true.)

      write (u, "(A)")
      write (u, "(A)")   "* Contents of mci_instance:"
      write (u, "(A)")

      call mci_instance%write (u, .true.)

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: mci_vamp_1"

    end subroutine mci_vamp_1
```

Multiple iterations

Construct an integrator and use it for a one-dimensional sampler. Integrate
with five iterations without grid adaptation.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_2, "mci_vamp_2", &
         "multiple iterations", &
         u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_2 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")   "* Test output: mci_vamp_2"
      write (u, "(A)")   "*   Purpose: integrate function in one dimension &
           &(single channel)"

      write (u, "(A)")
      write (u, "(A)")   "* Initialize integrator, sampler, instance"
```

```
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 1)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_1_t :: sampler)
  select type (sampler)
  type is (test_sampler_1_t)
     sampler%mode = 2
  end select
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
  write (u, "(A)")

  select type (mci)
  type is (mci_vamp_t)
     call mci%add_pass (adapt_grids = .false.)
  end select
  call mci%integrate (mci_instance, sampler, 3, 100)
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Contents of mci_instance:"
  write (u, "(A)")

  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call mci_instance%final ()
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_vamp_2"

end subroutine mci_vamp_2
```

Grid adaptation

Construct an integrator and use it for a one-dimensional sampler. Integrate with three iterations and in-between grid adaptations.

⟨*MCI vamp: execute tests*⟩+≡
```
call test (mci_vamp_3, "mci_vamp_3", &
     "grid adaptation", &
     u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine mci_vamp_3 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_3"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension &
       &(single channel)"
  write (u, "(A)")  "*           and adapt grid"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 1)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_1_t :: sampler)
  select type (sampler)
  type is (test_sampler_1_t)
     sampler%mode = 2
  end select
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
  write (u, "(A)")

  select type (mci)
  type is (mci_vamp_t)
```

1703

```
      call mci%add_pass (adapt_grids = .true.)
    end select
    call mci%integrate (mci_instance, sampler, 3, 100)
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_3"

  end subroutine mci_vamp_3
```

Two-dimensional integral

Construct an integrator and use it for a two-dimensional sampler. Integrate
with three iterations and in-between grid adaptations.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_4, "mci_vamp_4", &
        "two-dimensional integration", &
        u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
  subroutine mci_vamp_4 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_4"
    write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
        &(single channel)"
    write (u, "(A)")  "*             and adapt grid"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 1)
    select type (mci)
    type is (mci_vamp_t)
```

1704

```
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
   end select

   allocate (rng_tao_t :: rng)
   call rng%init ()
   call mci%import_rng (rng)

   call mci%allocate_instance (mci_instance)
   call mci_instance%init (mci)

   allocate (test_sampler_1_t :: sampler)
   select type (sampler)
   type is (test_sampler_1_t)
      sampler%mode = 3
   end select
   call sampler%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
   write (u, "(A)")

   select type (mci)
   type is (mci_vamp_t)
      call mci%add_pass (adapt_grids = .true.)
   end select
   call mci%integrate (mci_instance, sampler, 3, 1000)
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Contents of mci_instance:"
   write (u, "(A)")

   call mci_instance%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci_instance%final ()
   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_vamp_4"

 end subroutine mci_vamp_4
```

Two-channel integral

Construct an integrator and use it for a two-dimensional sampler with two channels.

Integrate with three iterations and in-between grid adaptations.

⟨*MCI vamp: execute tests*⟩+≡

```
      call test (mci_vamp_5, "mci_vamp_5", &
           "two-dimensional integration", &
           u, results)
```
⟨*MCI vamp: tests*⟩+≡
```
     subroutine mci_vamp_5 (u)
       integer, intent(in) :: u
       type(grid_parameters_t) :: grid_par
       class(mci_t), allocatable, target :: mci
       class(mci_instance_t), pointer :: mci_instance => null ()
       class(mci_sampler_t), allocatable :: sampler
       class(rng_t), allocatable :: rng

       write (u, "(A)")  "* Test output: mci_vamp_5"
       write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
           &(two channels)"
       write (u, "(A)")  "*              and adapt grid"

       write (u, "(A)")
       write (u, "(A)")  "* Initialize integrator, sampler, instance"
       write (u, "(A)")

       allocate (mci_vamp_t :: mci)
       call mci%set_dimensions (2, 2)
       select type (mci)
       type is (mci_vamp_t)
          grid_par%stratified = .false.
          grid_par%use_vamp_equivalences = .false.
          call mci%set_grid_parameters (grid_par)
       end select

       allocate (rng_tao_t :: rng)
       call rng%init ()
       call mci%import_rng (rng)

       call mci%allocate_instance (mci_instance)
       call mci_instance%init (mci)

       allocate (test_sampler_2_t :: sampler)
       call sampler%write (u)

       write (u, "(A)")
       write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
       write (u, "(A)")

       select type (mci)
       type is (mci_vamp_t)
          call mci%add_pass (adapt_grids = .true.)
       end select
       call mci%integrate (mci_instance, sampler, 3, 1000)
       call mci%write (u)

       write (u, "(A)")
       write (u, "(A)")  "* Contents of mci_instance:"
       write (u, "(A)")
```

1706

```
      call mci_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_vamp_5"

    end subroutine mci_vamp_5
```

Weight adaptation

Construct an integrator and use it for a one-dimensional sampler with two channels.

Integrate with three iterations and in-between weight adaptations.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_6, "mci_vamp_6", &
        "weight adaptation", &
        u, results)
```
⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_6 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_vamp_6"
      write (u, "(A)")  "*   Purpose: integrate function in one dimension &
          &(two channels)"
      write (u, "(A)")  "*            and adapt weights"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")

      allocate (mci_vamp_t :: mci)
      call mci%set_dimensions (1, 2)
      select type (mci)
      type is (mci_vamp_t)
         grid_par%stratified = .false.
         grid_par%use_vamp_equivalences = .false.
         call mci%set_grid_parameters (grid_par)
      end select

      allocate (rng_tao_t :: rng)
      call rng%init ()
```

```
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.9_default
   sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()
deallocate (mci_instance)
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Re-initialize with chained channels"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
call mci%declare_chains ([1,1])
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
```

```
        call mci%import_rng (rng)

        call mci%allocate_instance (mci_instance)
        call mci_instance%init (mci)

        write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
        write (u, "(A)")

        select type (mci)
        type is (mci_vamp_t)
           call mci%add_pass (adapt_weights = .true.)
        end select
        call mci%integrate (mci_instance, sampler, 3, 1000)
        call mci%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Contents of mci_instance:"
        write (u, "(A)")

        call mci_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call mci_instance%final ()
        call mci%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: mci_vamp_6"

      end subroutine mci_vamp_6
```

Equivalences

Construct an integrator and use it for a one-dimensional sampler with two channels.

Integrate with three iterations and in-between grid adaptations. Apply an equivalence between the two channels, so the binning of the two channels is forced to coincide. Compare this with the behavior without equivalences.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_7, "mci_vamp_7", &
         "use channel equivalences", &
         u, results)
```
⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_7 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      type(phs_channel_t), dimension(:), allocatable :: channel
      class(rng_t), allocatable :: rng
```

1709

```
real(default), dimension(:,:), allocatable :: x
integer :: u_grid, iostat, i, div, ch
character(16) :: buffer

write (u, "(A)")  "* Test output: mci_vamp_7"
write (u, "(A)")  "*   Purpose: check effect of channel equivalences"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.7_default
   sampler%b = 0.3_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 2 and n_calls = 1000, &
     &adapt grids"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 2, 1000)

call mci%write (u)

write (u, "(A)")
write (u, "(A)") "* Write grids and extract binning"
write (u, "(A)")

u_grid = free_unit ()
open (u_grid, status = "scratch", action = "readwrite")
```

1710

```
select type (mci_instance)
type is (mci_vamp_instance_t)
   call vamp_write_grids (mci_instance%grids, u_grid)
end select
rewind (u_grid)
allocate (x (0:20, 2))
do div = 1, 2
   FIND_BINS1: do
      read (u_grid, "(A)")  buffer
      if (trim (adjustl (buffer)) == "begin d%x") then
         do
            read (u_grid, *, iostat = iostat)  i, x(i,div)
            if (iostat /= 0)  exit FIND_BINS1
         end do
      end if
   end do FIND_BINS1
end do
close (u_grid)

write (u, "(1x,A,L1)")  "Equal binning in both channels = ", &
     all (x(:,1) == x(:,2))
deallocate (x)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()
deallocate (mci_instance)
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Re-initialize integrator, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .true.
   call mci%set_grid_parameters (grid_par)
end select

write (u, "(A)")  "* Define equivalences"
write (u, "(A)")

allocate (channel (2))
do ch = 1, 2
   allocate (channel(ch)%eq (2))
   do i = 1, 2
      associate (eq => channel(ch)%eq(i))
         call eq%init (1)
         eq%c = i
```

```
         eq%perm = [1]
         eq%mode = [0]
       end associate
   end do
   write (u, "(1x,I0,':')", advance = "no")  ch
   call channel(ch)%write (u)
end do
call mci%declare_equivalences (channel, dim_offset = 0)

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 2 and n_calls = 1000, &
     &adapt grids"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 2, 1000)

call mci%write (u)

write (u, "(A)")
write (u, "(A)") "* Write grids and extract binning"
write (u, "(A)")

u_grid = free_unit ()
open (u_grid, status = "scratch", action = "readwrite")
select type (mci_instance)
type is (mci_vamp_instance_t)
   call vamp_write_grids (mci_instance%grids, u_grid)
end select
rewind (u_grid)
allocate (x (0:20, 2))
do div = 1, 2
   FIND_BINS2: do
      read (u_grid, "(A)")  buffer
      if (trim (adjustl (buffer)) == "begin d%x") then
         do
            read (u_grid, *, iostat = iostat)  i, x(i,div)
            if (iostat /= 0)  exit FIND_BINS2
         end do
      end if
   end do FIND_BINS2
end do
close (u_grid)
```

```
      write (u, "(1x,A,L1)")  "Equal binning in both channels = ", &
           all (x(:,1) == x(:,2))
      deallocate (x)


      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_vamp_7"

    end subroutine mci_vamp_7
```

Multiple passes

Integrate with three passes and different settings for weight and grid adaptation.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_8, "mci_vamp_8", &
         "integration passes", &
         u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_8 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_vamp_8"
      write (u, "(A)")  "*   Purpose: integrate function in one dimension &
           &(two channels)"
      write (u, "(A)")  "*            in three passes"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")

      allocate (mci_vamp_t :: mci)
      call mci%set_dimensions (1, 2)
      select type (mci)
      type is (mci_vamp_t)
         grid_par%stratified = .false.
         grid_par%use_vamp_equivalences = .false.
         call mci%set_grid_parameters (grid_par)
      end select

      allocate (rng_tao_t :: rng)
      call rng%init ()
```

1713

```
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.9_default
   sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with grid and weight adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true., adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with grid adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate without adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass ()
```

```
      end select
      call mci%integrate (mci_instance, sampler, 3, 1000)
      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Contents of mci_instance:"
      write (u, "(A)")

      call mci_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_vamp_8"

    end subroutine mci_vamp_8
```

Weighted events

Construct an integrator and use it for a two-dimensional sampler with two
channels. Integrate and generate a weighted event.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_9, "mci_vamp_9", &
         "weighted event", &
         u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_9 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_vamp_9"
      write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
           &(two channels)"
      write (u, "(A)")  "*            and generate a weighted event"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")

      allocate (mci_vamp_t :: mci)
      call mci%set_dimensions (2, 2)
      select type (mci)
      type is (mci_vamp_t)
         grid_par%stratified = .false.
```

```
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_2_t :: sampler)
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
    write (u, "(A)")

    call mci%add_pass ()
    call mci%integrate (mci_instance, sampler, 1, 1000)
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate a weighted event"
    write (u, "(A)")

    call mci_instance%init_simulation ()
    call mci%generate_weighted_event (mci_instance, sampler)

    write (u, "(1x,A)")  "MCI instance:"
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final_simulation ()
    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_9"

  end subroutine mci_vamp_9
```

Grids I/O

Construct an integrator and allocate grids. Write grids to file, read them in
again and compare.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_10, "mci_vamp_10", &
         "grids I/O", &
         u, results)
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_10 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  type(string_t) :: file1, file2
  character(80) :: buffer1, buffer2
  integer :: u1, u2, iostat1, iostat2
  logical :: equal, success

  write (u, "(A)")  "* Test output: mci_vamp_10"
  write (u, "(A)")  "*   Purpose: write and read VAMP grids"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%stratified = .false.
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  mci%md5sum = "1234567890abcdef1234567890abcdef"

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_2_t :: sampler)
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
  write (u, "(A)")

  call mci%add_pass ()
  call mci%integrate (mci_instance, sampler, 1, 1000)

  write (u, "(A)")  "* Write grids to file"
  write (u, "(A)")

  file1 = "mci_vamp_10.1"
  select type (mci)
  type is (mci_vamp_t)
```

```
      call mci%set_grid_filename (file1)
      call mci%write_grids (mci_instance)
end select

call mci_instance%final ()
call mci%final ()
deallocate (mci)

write (u, "(A)")  "* Read grids from file"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

mci%md5sum = "1234567890abcdef1234567890abcdef"

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

select type (mci)
type is (mci_vamp_t)
   call mci%set_grid_filename (file1)
   call mci%add_pass ()
   call mci%current_pass%configure (1, 1000, &
        mci%min_calls, &
        mci%grid_par%min_bins, mci%grid_par%max_bins, &
        mci%grid_par%min_calls_per_channel * mci%n_channel)
   call mci%read_grids_header (success)
   call mci%compute_md5sum ()
   call mci%read_grids_data (mci_instance, read_integrals = .true.)
end select
write (u, "(1x,A,L1)")  "success = ", success

write (u, "(A)")
write (u, "(A)")  "* Write grids again"
write (u, "(A)")

file2 = "mci_vamp_10.2"
select type (mci)
type is (mci_vamp_t)
   call mci%set_grid_filename (file2)
   call mci%write_grids (mci_instance)
end select

u1 = free_unit ()
open (u1, file = char (file1) // ".vg", action = "read", status = "old")
```

```
            u2 = free_unit ()
            open (u2, file = char (file2) // ".vg", action = "read", status = "old")

            equal = .true.
            iostat1 = 0
            iostat2 = 0
            do while (equal .and. iostat1 == 0 .and. iostat2 == 0)
               read (u1, "(A)", iostat = iostat1)  buffer1
               read (u2, "(A)", iostat = iostat2)  buffer2
               equal = buffer1 == buffer2 .and. iostat1 == iostat2
            end do
            close (u1)
            close (u2)

            if (equal) then
               write (u, "(1x,A)")  "Success: grid files are identical"
            else
               write (u, "(1x,A)")  "Failure: grid files differ"
            end if

            write (u, "(A)")
            write (u, "(A)")  "* Cleanup"

            call mci_instance%final ()
            call mci%final ()

            write (u, "(A)")
            write (u, "(A)")  "* Test output end: mci_vamp_10"

         end subroutine mci_vamp_10
```

Weighted events

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate, write grids, and generate a weighted event using the grids from file.

⟨*MCI vamp: execute tests*⟩+≡
```
      call test (mci_vamp_11, "mci_vamp_11", &
           "weighted events with grid I/O", &
           u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
      subroutine mci_vamp_11 (u)
        integer, intent(in) :: u
        type(grid_parameters_t) :: grid_par
        class(mci_t), allocatable, target :: mci
        class(mci_instance_t), pointer :: mci_instance => null ()
        class(mci_sampler_t), allocatable :: sampler
        class(rng_t), allocatable :: rng

        write (u, "(A)")  "* Test output: mci_vamp_11"
        write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
             &(two channels)"
```

1719

```
    write (u, "(A)")  "*             and generate a weighted event"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_vamp_t)
       grid_par%stratified = .false.
       grid_par%use_vamp_equivalences = .false.
       call mci%set_grid_parameters (grid_par)
       call mci%set_grid_filename (var_str ("mci_vamp_11"))
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_2_t :: sampler)

    write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
    write (u, "(A)")

    call mci%add_pass ()
    call mci%integrate (mci_instance, sampler, 1, 1000)

    write (u, "(A)")  "* Reset instance"
    write (u, "(A)")

    call mci_instance%final ()
    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Generate a weighted event"
    write (u, "(A)")

    call mci_instance%init_simulation ()
    call mci%generate_weighted_event (mci_instance, sampler)

    write (u, "(A)")  "* Cleanup"

    call mci_instance%final_simulation ()
    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_11"

  end subroutine mci_vamp_11
```

1720

Weighted events

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate, write grids, and generate a weighted event using the grids from file.

⟨*MCI vamp: execute tests*⟩+≡
```
call test (mci_vamp_12, "mci_vamp_12", &
     "unweighted events with grid I/O", &
     u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine mci_vamp_12 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_12"
  write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
      &(two channels)"
  write (u, "(A)")  "*             and generate an unweighted event"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%stratified = .false.
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
     call mci%set_grid_filename (var_str ("mci_vamp_12"))
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_2_t :: sampler)

  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
  write (u, "(A)")

  call mci%add_pass ()
  call mci%integrate (mci_instance, sampler, 1, 1000)
```

```
          write (u, "(A)")  "* Reset instance"
          write (u, "(A)")

          call mci_instance%final ()
          call mci%allocate_instance (mci_instance)
          call mci_instance%init (mci)

          write (u, "(A)")  "* Generate an unweighted event"
          write (u, "(A)")

          call mci_instance%init_simulation ()
          call mci%generate_unweighted_event (mci_instance, sampler)

          write (u, "(1x,A)")  "MCI instance:"
          call mci_instance%write (u)

          write (u, "(A)")
          write (u, "(A)")  "* Cleanup"

          call mci_instance%final_simulation ()
          call mci_instance%final ()
          call mci%final ()

          write (u, "(A)")
          write (u, "(A)")  "* Test output end: mci_vamp_12"

        end subroutine mci_vamp_12
```

Update integration results

Compare two `mci` objects; match the two and update the first if successful.

⟨*MCI vamp: execute tests*⟩+≡
```
      call test (mci_vamp_13, "mci_vamp_13", &
           "updating integration results", &
           u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
      subroutine mci_vamp_13 (u)
        integer, intent(in) :: u
        type(grid_parameters_t) :: grid_par
        class(mci_t), allocatable, target :: mci, mci_ref
        logical :: success

        write (u, "(A)")  "* Test output: mci_vamp_13"
        write (u, "(A)")  "*   Purpose: match and update integrators"

        write (u, "(A)")
        write (u, "(A)")  "* Initialize integrator with no passes"
        write (u, "(A)")

        allocate (mci_vamp_t :: mci)
        call mci%set_dimensions (2, 2)
```

```
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
end select
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize reference"
write (u, "(A)")

allocate (mci_vamp_t :: mci_ref)
call mci_ref%set_dimensions (2, 2)
select type (mci_ref)
type is (mci_vamp_t)
   call mci_ref%set_grid_parameters (grid_par)
end select

select type (mci_ref)
type is (mci_vamp_t)
   call mci_ref%add_pass (adapt_grids = .true.)
   call mci_ref%current_pass%configure (2, 1000, 0, 1, 5, 0)
   mci_ref%current_pass%calls = [77, 77]
   mci_ref%current_pass%integral = [1.23_default, 3.45_default]
   mci_ref%current_pass%error = [0.23_default, 0.45_default]
   mci_ref%current_pass%efficiency = [0.1_default, 0.6_default]
   mci_ref%current_pass%integral_defined = .true.

   call mci_ref%add_pass ()
   call mci_ref%current_pass%configure (2, 2000, 0, 1, 7, 0)
   mci_ref%current_pass%calls = [99, 0]
   mci_ref%current_pass%integral = [7.89_default, 0._default]
   mci_ref%current_pass%error = [0.89_default, 0._default]
   mci_ref%current_pass%efficiency = [0.86_default, 0._default]
   mci_ref%current_pass%integral_defined = .true.
end select

call mci_ref%write (u)

write (u, "(A)")
write (u, "(A)")  "* Update integrator (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
```

1723

```
write (u, "(A)")  "* Add pass to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
   call mci%current_pass%configure (2, 1000, 0, 1, 5, 0)
   mci%current_pass%calls = [77, 77]
   mci%current_pass%integral = [1.23_default, 3.45_default]
   mci%current_pass%error = [0.23_default, 0.45_default]
   mci%current_pass%efficiency = [0.1_default, 0.6_default]
   mci%current_pass%integral_defined = .true.
end select

write (u, "(A)")  "* Update integrator (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Add pass to integrator, wrong parameters"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass ()
   call mci%current_pass%configure (2, 1000, 0, 1, 7, 0)
end select

write (u, "(A)")  "* Update integrator (should fail)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Reset and add passes to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
```

```
   call mci%reset ()
   call mci%add_pass (adapt_grids = .true.)
   call mci%current_pass%configure (2, 1000, 0, 1, 5, 0)
   mci%current_pass%calls = [77, 77]
   mci%current_pass%integral = [1.23_default, 3.45_default]
   mci%current_pass%error = [0.23_default, 0.45_default]
   mci%current_pass%efficiency = [0.1_default, 0.6_default]
   mci%current_pass%integral_defined = .true.

   call mci%add_pass ()
   call mci%current_pass%configure (2, 2000, 0, 1, 7, 0)
end select

write (u, "(A)")  "* Update integrator (should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Update again (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Add extra result to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   mci%current_pass%calls(2) = 1234
end select

write (u, "(A)")  "* Update integrator (should fail)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select
```

```
      write (u, "(1x,A,L1)")  "success = ", success
      write (u, "(A)")
      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call mci%final ()
      call mci_ref%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_vamp_13"

   end subroutine mci_vamp_13
```

Accuracy Goal

Integrate with multiple iterations. Skip iterations once an accuracy goal has been reached.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_14, "mci_vamp_14", &
         "accuracy goal", &
         u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_14 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_vamp_14"
      write (u, "(A)")  "*   Purpose: integrate function in one dimension &
           &(single channel)"
      write (u, "(A)")  "*            and check accuracy goal"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")

      allocate (mci_vamp_t :: mci)
      call mci%set_dimensions (1, 1)
      select type (mci)
      type is (mci_vamp_t)
         grid_par%use_vamp_equivalences = .false.
         grid_par%accuracy_goal = 5E-2_default
         call mci%set_grid_parameters (grid_par)
      end select

      allocate (rng_tao_t :: rng)
```

```
      call rng%init ()
      call mci%import_rng (rng)

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      allocate (test_sampler_1_t :: sampler)
      select type (sampler)
      type is (test_sampler_1_t)
         sampler%mode = 2
      end select
      call sampler%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Integrate with n_it = 5 and n_calls = 100"
      write (u, "(A)")

      select type (mci)
      type is (mci_vamp_t)
         call mci%add_pass (adapt_grids = .true.)
      end select
      call mci%integrate (mci_instance, sampler, 5, 100)
      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call mci_instance%final ()
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_vamp_14"

    end subroutine mci_vamp_14
```

VAMP history

Integrate with three passes and different settings for weight and grid adaptation.
Then show the VAMP history.

⟨*MCI vamp: execute tests*⟩+≡
```
    call test (mci_vamp_15, "mci_vamp_15", &
         "VAMP history", &
         u, results)
```

⟨*MCI vamp: tests*⟩+≡
```
    subroutine mci_vamp_15 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      type(history_parameters_t) :: history_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng
```

```
write (u, "(A)")  "* Test output: mci_vamp_15"
write (u, "(A)")  "*   Purpose: integrate function in one dimension &
    &(two channels)"
write (u, "(A)")  "*            in three passes, show history"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

history_par%channel = .true.

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
   call mci%set_history_parameters (history_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.9_default
   sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Pass 1: grid and weight adaptation"

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true., adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)

write (u, "(A)")
write (u, "(A)")  "* Pass 2: grid adaptation"

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
```

1728

```
  write (u, "(A)")
  write (u, "(A)")  "* Pass 3: without adaptation"

  select type (mci)
  type is (mci_vamp_t)
     call mci%add_pass ()
  end select
  call mci%integrate (mci_instance, sampler, 3, 1000)

  write (u, "(A)")
  write (u, "(A)")  "* Contents of MCI record, with history"
  write (u, "(A)")

  call mci%write (u)
  select type (mci)
  type is (mci_vamp_t)
     call mci%write_history (u)
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call mci_instance%final ()
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_vamp_15"

end subroutine mci_vamp_15
```

Chapter 14

Process Libraries

This chapter consists of two modules which organize and interface the interaction matrix elements for a variety of elementary processes.

prclib_interfaces This module deals with matrix-element code which is accessible via external libraries (Fortran libraries or generic C-compatible libraries) and must either be generated by the program or provided by the user explicitly.

The module defines and uses an abstract type `prc_writer_t` and two abstract extensions, one for a Fortran module and one for a C-compatible library. The implementation provides the specific methods for writing the appropriate parts in external matrix element code.

prc_core_def This module defines the abstract types `prc_core_def_t` and `prc_driver_t`. The implementation of the former provides the configuration for processes of a certain class, while the latter accesses the corresponding matrix element, in particular those generated by the appropriate `prc_writer_t` object.

process_libraries This module combines the functionality of the previous module with the means for holding processes definitions (the internal counterpart of appropriate declarations in the user interface), for handling matrix elements which do not need external code, and for accessing the matrix elements by the procedures for matrix-element evaluation, integration and event generation.

test_me This module provides a test implementation for the abstract types in the `prc_core_def` module. The implementation is intended for self-tests of several later modules. The implementation is internal, i.e., no external code has is generated.

All data structures which are specific for a particular way of generating code or evaluating matrix element are kept abstract and thus generic. Later modules such as `prc_omega` provide implementations, in the form of type extensions for the various abstract types.

14.1 Process library interface

The module `prclib_interfaces` handles external matrix-element code.

14.1.1 Overview

The top-level data structure is the `prclib_driver_t` data type. The associated type-bound procedures deal with the generation of external code, compilation and linking, and accessing the active external library.

An object of type `prclib_driver_t` consists of the following parts:

1. Metadata that identify name and status of the library driver, etc.

2. An array of process records (`prclib_driver_record_t`), one for each external matrix element.

3. A record of type `dlaccess_t` which handles the operating-system part of linking a dynamically loadable library.

4. A collection of procedure pointers which have a counterpart in the external library interface. Given the unique identifier of a matrix element, the procedures retrieve generic matrix-element information such as the particle content and helicity combination tables. There is also a procedure which returns pointers to the more specific procedures that a matrix element provides, called *features*.

The process records of type `prclib_driver_record_t` handle the individual matrix elements. Each record identifies a process by name (`id`), names the physics model to be loaded for this process, lists the features that the associated matrix-element code provides, and holds a `writer` object which handles all operations that depend on the process type. The numbering of process records is identical to the numbering of matrix-element codes in the external library.

The writer object is of abstract type `prc_writer_t`. The module defines two basic, also abstract, extensions: `prc_writer_f_module_t` and `prc_writer_c_lib_t`. The first version is for matrix-element code that is available in form of Fortran modules. The writer contains type-bound procedures which create appropriate `use` directives and C-compatible wrapper functions for the given set of Fortran modules and their features. The second version is for matrix-element code that is available in form of a C-compatible library (this includes Fortran libraries with proper C bindings). The writer needs not write wrapper function, but explicit interface blocks for the matrix-element features.

Each matrix-element variant is encoded in an appropriate extension of `prc_writer_t`. For instance, O'MEGA matrix elements provide an implementation `omega_writer_t` which extends `prc_writer_f_module_t`.

14.1.2 Workflow

We expect that the functionality provided by this module is called in the following order:

1. The caller initializes the `prclib_driver_t` object and fills the array of `prclib_record_t` entries with the appropriate process data and process-specific writer objects.

2. It calls the `generate_makefile` method to set up an appropriate makefile in the current directory. The makefile will handle source generation, compilation and linking both for the individual matrix elements (unless this has to be done manually) and for the common external driver code which interfaces those matrix element.

3. The `generate_driver_code` writes the common driver as source code to file.

4. The methods `make_source`, `make_compile`, and `make_link` individually perform the corresponding steps in building the library. Wherever possible, they simply use the generated makefile. By calling `make`, we make sure that we can avoid unnecessary recompilation. For the compilation and linking steps, the makefile will employ `libtool`.

5. The `load` method loads the library procedures into the corresponding procedure pointers, using the `dlopen` mechanism via the `dlaccess` subobject.

14.1.3   The module

⟨`prclib_interfaces.f90`⟩≡
  ⟨*File header*⟩

  module prclib_interfaces

    use iso_c_binding !NODEP!
    use kinds !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use limits, only: TAB !NODEP!
    use unit_tests
    use os_interface

  ⟨*Standard module head*⟩

  ⟨*Prclib interfaces: public*⟩

  ⟨*Prclib interfaces: types*⟩

  ⟨*Prclib interfaces: interfaces*⟩

  ⟨*Prclib interfaces: test types*⟩

  contains

  ⟨*Prclib interfaces: procedures*⟩

  ⟨*Prclib interfaces: tests*⟩

  end module prclib_interfaces

14.1.4 Writers

External matrix element code provides externally visible procedures, which we denote as *features*. The features consist of informational subroutines and functions which are mandatory (universal features) and matrix-element specific subroutines and functions (specific features). The driver interfaces the generic features directly, while it returns the specific features in form of bind(C) procedure pointers to the caller. For instance, function `n_in` is generic, while the matrix matrix-element value itself is specific.

To implement these tasks, the driver needs `use` directives for Fortran module procedures, interface blocks for other external stuff, wrapper code, and Makefile snippets.

Generic writer

In the `prc_writer_t` data type, we collect the procedures which implement the writing tasks. The type is abstract. The concrete implementations are defined by an extension which is specific for the process type.

The MD5 sum stored here should be the MD5 checksum of the current process component, which can be calculated once the process is configured completely. It can be used by implementations which work with external files, such as `O'MEGA`.

⟨*Prclib interfaces: public*⟩≡
```
    public :: prc_writer_t
```

⟨*Prclib interfaces: types*⟩≡
```
    type, abstract :: prc_writer_t
      character(32) :: md5sum = ""
     contains
     ⟨Prclib interfaces: prc writer: TBP⟩
    end type prc_writer_t
```

In any case, it is useful to have a string representation of the writer type. This must be implemented by all extensions.

⟨*Prclib interfaces: prc writer: TBP*⟩≡
```
    procedure(get_const_string), nopass, deferred :: type_name
```

⟨*Prclib interfaces: interfaces*⟩≡
```
    abstract interface
      function get_const_string () result (string)
        import
        type(string_t) :: string
      end function get_const_string
    end interface
```

Return the name of a procedure that implements a given feature, as it is provided by the external matrix-element code. For a reasonable default, we take the feature name unchanged.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
    procedure, nopass :: get_procname => prc_writer_get_procname
```

⟨*Prclib interfaces: procedures*⟩≡
```
function prc_writer_get_procname (feature) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: feature
  name = feature
end function prc_writer_get_procname
```

Return the name of a procedure that implements a given feature with the bind(C) property, so it can be accessed via a C procedure pointer and handled by dlopen. We need this for all special features of a matrix element, since the interface has to return a C function pointer for it. For a default implementation, we prefix the external procedure name by the process ID.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure :: get_c_procname => prc_writer_get_c_procname
```

⟨*Prclib interfaces: procedures*⟩+≡
```
function prc_writer_get_c_procname (writer, id, feature) result (name)
  class(prc_writer_t), intent(in) :: writer
  type(string_t), intent(in) :: id, feature
  type(string_t) :: name
  name = id // "_" // feature
end function prc_writer_get_c_procname
```

Common signature of code-writing procedures. The procedure may use the process ID, and the feature name. (Not necessarily all of them.)

⟨*Prclib interfaces: interfaces*⟩+≡
```
abstract interface
  subroutine write_code_file (writer, id)
    import
    class(prc_writer_t), intent(in) :: writer
    type(string_t), intent(in) :: id
  end subroutine write_code_file
end interface

abstract interface
  subroutine write_code (writer, unit, id)
    import
    class(prc_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
  end subroutine write_code
end interface

abstract interface
  subroutine write_code_os (writer, unit, id, os_data, testflag)
    import
    class(prc_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: testflag
  end subroutine write_code_os
end interface
```

1734

```
abstract interface
   subroutine write_feature_code (writer, unit, id, feature)
      import
      class(prc_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
   end subroutine write_feature_code
end interface
```

There must be a procedure which writes an interface block for a given feature. If the external matrix element is implemented as a Fortran module, this is required only for the specific features which are returned as procedure pointers.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure(write_feature_code), deferred :: write_interface
```

There must also be a procedure which writes Makefile code which is specific for the current process, but not the feature.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure(write_code_os), deferred :: write_makefile_code
```

This procedure writes code process-specific source-code file (which need not be Fortran). It may be a no-op, if the source code is generated by Make instead.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure(write_code_file), deferred :: write_source_code
```

Writer for Fortran-module matrix elements

If the matrix element is available as a Fortran module, we have specific requirements: (i) the features are imported via `use` directives, (ii) the specific features require bind(C) wrappers.

The type is still abstract, all methods must be implemented explicitly for a specific matrix-element variant.

⟨*Prclib interfaces: public*⟩+≡
```
public :: prc_writer_f_module_t
```

⟨*Prclib interfaces: types*⟩+≡
```
type, extends (prc_writer_t), abstract :: prc_writer_f_module_t
 contains
   ⟨Prclib interfaces: prc writer f module: TBP⟩
end type prc_writer_f_module_t
```

Return the name of the Fortran module. As a default implementation, we take the process ID unchanged.

⟨*Prclib interfaces: prc writer f module: TBP*⟩≡
```
procedure, nopass :: get_module_name => prc_writer_get_module_name
```

⟨*Prclib interfaces: procedures*⟩+≡
```
function prc_writer_get_module_name (id) result (name)
   type(string_t) :: name
   type(string_t), intent(in) :: id
   name = id
end function prc_writer_get_module_name
```

Write a `use` directive that associates the driver reference with the procedure in the matrix element code. By default, we use the C name for this.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
procedure :: write_use_line => prc_writer_write_use_line
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_write_use_line (writer, unit, id, feature)
  class(prc_writer_f_module_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t) :: id, feature
  write (unit, "(2x,9A)")  "use ", char (writer%get_module_name (id)), &
       ", only: ", char (writer%get_c_procname (id, feature)), &
       " => ", char (writer%get_procname (feature))
end subroutine prc_writer_write_use_line
```

Write a wrapper routine for a feature. This also associates a C name the module procedure. The details depend on the writer variant.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
procedure(prc_write_wrapper), deferred :: write_wrapper
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
abstract interface
   subroutine prc_write_wrapper (writer, unit, id, feature)
     import
     class(prc_writer_f_module_t), intent(in) :: writer
     integer, intent(in) :: unit
     type(string_t), intent(in) :: id, feature
   end subroutine prc_write_wrapper
end interface
```

This is used for testing only: initialize the writer with a specific MD5 sum string.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure :: init_test => prc_writer_init_test
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_init_test (writer)
  class(prc_writer_t), intent(out) :: writer
  writer%md5sum = "1234567890abcdef1234567890abcdef"
end subroutine prc_writer_init_test
```

Writer for C-library matrix elements

This applies if the matrix element is available as a C library or a Fortran library with bind(C) compatible interface. We can use the basic version.

The type is still abstract, all methods must be implemented explicitly for a specific matrix-element variant.

⟨*Prclib interfaces: types*⟩+≡
```
type, extends (prc_writer_t), abstract :: prc_writer_c_lib_t
 contains
 ⟨Prclib interfaces: prc writer c lib: TBP⟩
end type prc_writer_c_lib_t
```

14.1.5 Process records in the library driver

A process record holds the process (component) `ID`, the physics `model_name`, and the array of `feature`s that are implemented by the corresponding matrix element code.

The `writer` component holds procedures. The procedures write source code for the current record, either for the driver or for the Makefile.

⟨*Prclib interfaces: types*⟩+≡
```
type :: prclib_driver_record_t
  type(string_t) :: id
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: feature
  class(prc_writer_t), pointer :: writer => null ()
 contains
 ⟨Prclib interfaces: prclib driver record: TBP⟩
 end type prclib_driver_record_t
```

Output routine. We indent the output, so it smoothly integrates into the output routine for the whole driver.

Note: the pointer `writer` is introduced as a workaround for a NAG compiler bug.

⟨*Prclib interfaces: prclib driver record: TBP*⟩≡
```
procedure :: write => prclib_driver_record_write
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prclib_driver_record_write (object, unit)
  class(prclib_driver_record_t), intent(in) :: object
  integer, intent(in) :: unit
  integer :: j
  class(prc_writer_t), pointer :: writer
  write (unit, "(3x,A,2x,'[',A,']')")  &
      char (object%id), char (object%model_name)
  if (allocated (object%feature)) then
     writer => object%writer
     write (unit, "(5x,A,A)", advance="no") &
         char (writer%type_name ()), ":"
     do j = 1, size (object%feature)
        write (unit, "(1x,A)", advance="no") &
            char (object%feature(j))
     end do
     write (unit, *)
  end if
end subroutine prclib_driver_record_write
```

Get the C procedure name for a feature.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
procedure :: get_c_procname => prclib_driver_record_get_c_procname
```

⟨*Prclib interfaces: procedures*⟩+≡
```
function prclib_driver_record_get_c_procname (record, feature) result (name)
  type(string_t) :: name
  class(prclib_driver_record_t), intent(in) :: record
  type(string_t), intent(in) :: feature
```

```
      name = record%writer%get_c_procname (record%id, feature)
    end function prclib_driver_record_get_c_procname
```

Write a USE directive for a given feature. Applies only if the record corresponds to a Fortran module.

⟨Prclib interfaces: prclib driver record: TBP⟩+≡
```
    procedure :: write_use_line => prclib_driver_record_write_use_line
```

⟨Prclib interfaces: procedures⟩+≡
```
    subroutine prclib_driver_record_write_use_line (record, unit, feature)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      type(string_t), intent(in) :: feature
      select type (writer => record%writer)
      class is (prc_writer_f_module_t)
         call writer%write_use_line (unit, record%id, feature)
      end select
    end subroutine prclib_driver_record_write_use_line
```

The alternative: write an interface block for a given feature, unless the record corresponds to a Fortran module.

⟨Prclib interfaces: prclib driver record: TBP⟩+≡
```
    procedure :: write_interface => prclib_driver_record_write_interface
```

⟨Prclib interfaces: procedures⟩+≡
```
    subroutine prclib_driver_record_write_interface (record, unit, feature)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      type(string_t), intent(in) :: feature
      select type (writer => record%writer)
      class is (prc_writer_f_module_t)
      class default
         call writer%write_interface (unit, record%id, feature)
      end select
    end subroutine prclib_driver_record_write_interface
```

Write all special feature interfaces for the current record. Do this for all process variants.

⟨Prclib interfaces: prclib driver record: TBP⟩+≡
```
    procedure :: write_interfaces => prclib_driver_record_write_interfaces
```

⟨Prclib interfaces: procedures⟩+≡
```
    subroutine prclib_driver_record_write_interfaces (record, unit)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      integer :: i
      do i = 1, size (record%feature)
         call record%writer%write_interface (unit, record%id, record%feature(i))
      end do
    end subroutine prclib_driver_record_write_interfaces
```

Write the wrapper routines for this record, if it corresponds to a Fortran module.

⟨Prclib interfaces: prclib driver record: TBP⟩+≡
```
    procedure :: write_wrappers => prclib_driver_record_write_wrappers
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_record_write_wrappers (record, unit)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      integer :: i
      select type (writer => record%writer)
      class is (prc_writer_f_module_t)
         do i = 1, size (record%feature)
            call writer%write_wrapper (unit, record%id, record%feature(i))
         end do
      end select
    end subroutine prclib_driver_record_write_wrappers
```

Write the Makefile code for this record.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
    procedure :: write_makefile_code => prclib_driver_record_write_makefile_code
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_record_write_makefile_code &
         (record, unit, os_data, testflag)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: testflag
      call record%writer%write_makefile_code (unit, record%id, os_data, testflag)
    end subroutine prclib_driver_record_write_makefile_code
```

Write source-code files for this record. (Does nothing if the source code is handled by Makefile rules.)

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
    procedure :: write_source_code => prclib_driver_record_write_source_code
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_record_write_source_code (record)
      class(prclib_driver_record_t), intent(in) :: record
      call record%writer%write_source_code (record%id)
    end subroutine prclib_driver_record_write_source_code
```

14.1.6 The process library driver object

A `prclib_driver_t` object provides the interface to external matrix element code. The code is provided by an external library which is either statically or dynamically linked.

The dynamic and static versions of the library are two different implementations of the abstract base type.

The `basename` identifies the library, both by file names and by Fortran variable names.

The `loaded` flag becomes true once all procedure pointers to the matrix element have been assigned.

For a dynamical external library, the communication proceeds via a `dlaccess` object.

`n_processes` is the number of external process code components that are referenced by this library. The code is addressed by index (`i_lib` in the process library entry above). This number should be equal to the number returned by `get_n_prc`.

For each external process, there is a separate `record` which holds the data that are needed for the driver parts which are specific for a given process component. The actual pointers for the loaded library will be assigned elsewhere.

The remainder is a collection of procedure pointers, which can be assigned once all external code has been compiled and linked. The procedure pointers all take a process component code index as an argument. Most return information about the process component that should match the process definition. The `get_fptr` procedures return a function pointer, which is the actual means to compute matrix elements or retrieve associated data.

Finally, the `unload_hook` and `reload_hook` pointers allow for the insertion of additional code when a library is loaded.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prclib_driver_t
```

⟨*Prclib interfaces: types*⟩+≡
```
    type, abstract :: prclib_driver_t
      type(string_t) :: basename
      character(32) :: md5sum = ""
      logical :: loaded = .false.
      type(string_t) :: libname
      type(string_t) :: modellibs_ldflags
      integer :: n_processes = 0
      type(prclib_driver_record_t), dimension(:), allocatable :: record
      procedure(prc_get_n_processes), nopass, pointer :: &
           get_n_processes => null ()
      procedure(prc_get_stringptr), nopass, pointer :: &
           get_process_id_ptr => null ()
      procedure(prc_get_stringptr), nopass, pointer :: &
           get_model_name_ptr => null ()
      procedure(prc_get_stringptr), nopass, pointer :: &
           get_md5sum_ptr => null ()
      procedure(prc_get_log), nopass, pointer :: &
           get_openmp_status => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_in  => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_out => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_flv => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_hel => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_col => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_cin => null ()
      procedure(prc_get_int), nopass, pointer :: get_n_cf  => null ()
      procedure(prc_set_int_tab1), nopass, pointer :: &
           set_flv_state_ptr => null ()
      procedure(prc_set_int_tab1), nopass, pointer :: &
           set_hel_state_ptr => null ()
      procedure(prc_set_col_state), nopass, pointer :: &
           set_col_state_ptr => null ()
      procedure(prc_set_color_factors), nopass, pointer :: &
           set_color_factors_ptr => null ()
      procedure(prc_get_fptr), nopass, pointer :: get_fptr => null ()
```

```
  contains
  ⟨Prclib interfaces: prclib driver: TBP⟩
end type prclib_driver_t
```

This is the dynamic version. It contains a `dlaccess` object for communicating with the OS.

⟨*Prclib interfaces: types*⟩+≡
```
    type, extends (prclib_driver_t) :: prclib_driver_dynamic_t
      type(dlaccess_t) :: dlaccess
    contains
    ⟨Prclib interfaces: prclib driver dynamic: TBP⟩
    end type prclib_driver_dynamic_t
```

Print just the metadata. Procedure pointers cannot be printed.

⟨*Prclib interfaces: prclib driver: TBP*⟩≡
```
    procedure :: write => prclib_driver_write
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_write (object, unit, libpath)
      class(prclib_driver_t), intent(in) :: object
      integer, intent(in) :: unit
      logical, intent(in), optional :: libpath
      logical :: write_lib
      integer :: i
      write_lib = .true.
      if (present (libpath))  write_lib = libpath
      write (unit, "(1x,A,A)")  &
           "External matrix-element code library: ", char (object%basename)
      select type (object)
      type is (prclib_driver_dynamic_t)
        write (unit, "(3x,A,L1)")  "static   = F"
      class default
        write (unit, "(3x,A,L1)")  "static   = T"
      end select
      write (unit, "(3x,A,L1)")  "loaded   = ", object%loaded
      write (unit, "(3x,A,A,A)") "MD5 sum  = '", object%md5sum, "'"
      if (write_lib) then
        write (unit, "(3x,A,A,A)") "Mdl flags = '", &
             char (object%modellibs_ldflags), "'"
      end if
      select type (object)
      type is (prclib_driver_dynamic_t)
        write (unit, *)
        call object%dlaccess%write (unit)
      end select
      write (unit, *)
      if (allocated (object%record)) then
        write (unit, "(1x,A)")  "Matrix-element code entries:"
        do i = 1, object%n_processes
           call object%record(i)%write (unit)
        end do
      else
        write (unit, "(1x,A)")  "Matrix-element code entries: [undefined]"
```

```
        end if
      end subroutine prclib_driver_write
```

Allocate a library as either static or dynamic. For static libraries, the procedure defers control to an external procedure which knows about the available static libraries. By default, this procedure is empty, but when we build a stand-alone executable, we replace the dummy by an actual dispatcher for the available static libraries. If the static dispatcher was not successful, we allocate a dynamic library.

The default version of `dispatch_prclib_static` resides in the `prebuilt` section of the `WHIZARD` tree, in a separate library. It does nothing, but can be replaced by a different procedure that allocates a static library driver if requested by name.

Note: `intent(out)` for the `driver` argument segfaults with gfortran 4.7.

⟨*Prclib interfaces: public*⟩+≡
```
      public :: dispatch_prclib_driver
```

⟨*Prclib interfaces: procedures*⟩+≡
```
      subroutine dispatch_prclib_driver &
          (driver, basename, modellibs_ldflags)
        class(prclib_driver_t), intent(inout), allocatable :: driver
        type(string_t), intent(in) :: basename
        type(string_t), intent(in), optional :: modellibs_ldflags
        procedure(dispatch_prclib_driver) :: dispatch_prclib_static
        if (allocated (driver))  deallocate (driver)
        call dispatch_prclib_static (driver, basename)
        if (.not. allocated (driver)) then
           allocate (prclib_driver_dynamic_t :: driver)
        end if
        driver%basename = basename
        driver%modellibs_ldflags = modellibs_ldflags
      end subroutine dispatch_prclib_driver
```

Initialize the ID array and set `n_processes` accordingly.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
      procedure :: init => prclib_driver_init
```

⟨*Prclib interfaces: procedures*⟩+≡
```
      subroutine prclib_driver_init (driver, n_processes)
        class(prclib_driver_t), intent(inout) :: driver
        integer, intent(in) :: n_processes
        driver%n_processes = n_processes
        allocate (driver%record (n_processes))
      end subroutine prclib_driver_init
```

Set the MD5 sum. This is separate because the MD5 sum may be known only after initialization.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
      procedure :: set_md5sum => prclib_driver_set_md5sum
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_set_md5sum (driver, md5sum)
      class(prclib_driver_t), intent(inout) :: driver
      character(32), intent(in) :: md5sum
      driver%md5sum = md5sum
    end subroutine prclib_driver_set_md5sum
```

Set the process record for a specific library entry. If the index is zero, we do nothing.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: set_record => prclib_driver_set_record
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_set_record (driver, i, &
        id, model_name, features, writer)
      class(prclib_driver_t), intent(inout) :: driver
      integer, intent(in) :: i
      type(string_t), intent(in) :: id
      type(string_t), intent(in) :: model_name
      type(string_t), dimension(:), intent(in) :: features
      class(prc_writer_t), intent(in), pointer :: writer
      if (i > 0) then
         associate (record => driver%record(i))
           record%id = id
           record%model_name = model_name
           allocate (record%feature (size (features)))
           record%feature = features
           record%writer => writer
         end associate
      end if
    end subroutine prclib_driver_set_record
```

Write all USE directives for a given feature, scanning the array of processes. Only Fortran-module processes count. Then, write interface blocks for the remaining processes.

The `implicit none` statement must go in-between.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_interfaces => prclib_driver_write_interfaces
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_write_interfaces (driver, unit, feature)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: feature
      integer :: i
      do i = 1, driver%n_processes
         call driver%record(i)%write_use_line (unit, feature)
      end do
      write (unit, "(2x,9A)")  "implicit none"
      do i = 1, driver%n_processes
         call driver%record(i)%write_interface (unit, feature)
      end do
    end subroutine prclib_driver_write_interfaces
```

14.1.7 Write makefile

The makefile contains constant parts, parts that depend on the library name,
and parts that depend on the specific processes and their types.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: generate_makefile => prclib_driver_generate_makefile
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_generate_makefile (driver, unit, os_data, testflag)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: testflag
      integer :: i
      write (unit, "(A)")  "# WHIZARD: Makefile for process library '" &
            // char (driver%basename) // "'"
      write (unit, "(A)")  "# Automatically generated file, do not edit"
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# Integrity check (don't modify the following line!)"
      write (unit, "(A)")  "MD5SUM = '" // driver%md5sum // "'"
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# Library name"
      write (unit, "(A)")  "BASE = " // char (driver%basename)
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# Compiler"
      write (unit, "(A)")  "FC = " // char (os_data%fc)
      write (unit, "(A)")  "CC = " // char (os_data%cc)
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# Included libraries"
      write (unit, "(A)")  "FCINCL = " // char (os_data%whizard_includes)
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# Compiler flags"
      write (unit, "(A)")  "FCFLAGS = " // char (os_data%fcflags)
      write (unit, "(A)")  "FCFLAGS_PIC = " // char (os_data%fcflags_pic)
      write (unit, "(A)")  "CFLAGS = " // char (os_data%cflags)
      write (unit, "(A)")  "CFLAGS_PIC = " // char (os_data%cflags_pic)
      write (unit, "(A)")  "LDFLAGS = " // char (os_data%whizard_ldflags) &
            // " " // char (os_data%ldflags) // " " // &
            char (driver%modellibs_ldflags)
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# LaTeX setup"
      write (unit, "(A)")  "LATEX = " // char (os_data%latex)
      write (unit, "(A)")  "MPOST = " // char (os_data%mpost)
      write (unit, "(A)")  "DVIPS = " // char (os_data%dvips)
      write (unit, "(A)")  "PS2PDF = " // char (os_data%ps2pdf)
      write (unit, "(A)")  'TEX_FLAGS = "$$TEXINPUTS:' // &
            char(os_data%whizard_texpath) // '"'
      write (unit, "(A)")  'MP_FLAGS  = "$$MPINPUTS:' // &
            char(os_data%whizard_texpath) // '"'
      write (unit, "(A)")  ""
      write (unit, "(A)")  "# Libtool"
      write (unit, "(A)")  "LIBTOOL = " // char (os_data%whizard_libtool)
      write (unit, "(A)")  "FCOMPILE = $(LIBTOOL) --tag=FC --mode=compile"
      write (unit, "(A)")  "CCOMPILE = $(LIBTOOL) --tag=CC --mode=compile"
      write (unit, "(A)")  "LINK = $(LIBTOOL) --tag=FC --mode=link"
```

```
write (unit, "(A)")   ""
write (unit, "(A)")   "# Compile commands (default)"
write (unit, "(A)")   "LTFCOMPILE = $(FCOMPILE) $(FC) -c &
    &$(FCINCL) $(FCFLAGS) $(FCFLAGS_PIC)"
write (unit, "(A)")   "LTCCOMPILE = $(CCOMPILE) $(CC) -c &
    &$(CFLAGS) $(CFLAGS_PIC)"
write (unit, "(A)")   ""
write (unit, "(A)")   "# Default target"
write (unit, "(A)")   "all: link diags"
write (unit, "(A)")   ""
write (unit, "(A)")   "# Matrix-element code files"
do i = 1, size (driver%record)
   call driver%record(i)%write_makefile_code (unit, os_data, testflag)
end do
write (unit, "(A)")   ""
write (unit, "(A)")   "# Library driver"
write (unit, "(A)")   "$(BASE).lo: $(BASE).f90 $(OBJECTS)"
write (unit, "(A)")   TAB // "$(LTFCOMPILE) $<"
write (unit, "(A)")   ""
write (unit, "(A)")   "# Library"
write (unit, "(A)")   "$(BASE).la: $(BASE).lo $(OBJECTS)"
write (unit, "(A)")   TAB // "$(LINK) $(FC) -module -rpath /dev/null &
    &$(FCFLAGS) $(LDFLAGS) -o $(BASE).la $^"
write (unit, "(A)")   ""
write (unit, "(A)")   "# Main targets"
write (unit, "(A)")   "link: compile $(BASE).la"
write (unit, "(A)")   "compile: source $(OBJECTS) $(TEX_OBJECTS) $(BASE).lo"
write (unit, "(A)")   "compile_tex: $(TEX_OBJECTS)"
write (unit, "(A)")   "source: $(SOURCES) $(BASE).f90 $(TEX_SOURCES)"
write (unit, "(A)")   ".PHONY: link diags compile compile_tex source"
write (unit, "(A)")   ""
write (unit, "(A)")   "# Specific cleanup targets"
do i = 1, size (driver%record)
   write (unit, "(A)")   "clean-" // char (driver%record(i)%id) // ":"
   write (unit, "(A)")   ".PHONY: clean-" // char (driver%record(i)%id)
end do
write (unit, "(A)")   ""
write (unit, "(A)")   "# Generic cleanup targets"
write (unit, "(A)")   "clean-library:"
write (unit, "(A)")   TAB // "rm -f $(BASE).la"
write (unit, "(A)")   "clean-objects:"
write (unit, "(A)")   TAB // "rm -f $(BASE).lo $(BASE)_driver.mod &
    &$(CLEAN_OBJECTS)"
write (unit, "(A)")   "clean-source:"
write (unit, "(A)")   TAB // "rm -f $(CLEAN_SOURCES)"
write (unit, "(A)")   "clean-driver:"
write (unit, "(A)")   TAB // "rm -f $(BASE).f90"
write (unit, "(A)")   "clean-makefile:"
write (unit, "(A)")   TAB // "rm -f $(BASE).makefile"
write (unit, "(A)")   ".PHONY: clean-library clean-objects &
    &clean-source clean-driver clean-makefile"
write (unit, "(A)")   ""
write (unit, "(A)")   "clean: clean-library clean-objects clean-source"
write (unit, "(A)")   "distclean: clean clean-driver clean-makefile"
```

1745

```
      write (unit, "(A)")  ".PHONY: clean distclean"
    end subroutine prclib_driver_generate_makefile
```

14.1.8   Write driver file

This procedure writes the process library driver source code to the specified
output unit. The individual routines for writing source-code procedures are
given below.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: generate_driver_code => prclib_driver_generate_code
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_generate_code (driver, unit)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t) :: prefix
      integer :: i

      prefix = driver%basename // "_"

      write (unit, "(A)")  "! WHIZARD matrix-element code interface"
      write (unit, "(A)")  "!"
      write (unit, "(A)")  "! Automatically generated file, do not edit"
      call driver%write_module (unit, prefix)
      call driver%write_lib_md5sum_fun (unit, prefix)
      call driver%write_get_n_processes_fun (unit, prefix)
      call driver%write_get_process_id_fun (unit, prefix)
      call driver%write_get_model_name_fun (unit, prefix)
      call driver%write_get_md5sum_fun (unit, prefix)
      call driver%write_string_to_array_fun (unit, prefix)
      call driver%write_get_openmp_status_fun (unit, prefix)
      call driver%write_get_int_fun (unit, prefix, var_str ("n_in"))
      call driver%write_get_int_fun (unit, prefix, var_str ("n_out"))
      call driver%write_get_int_fun (unit, prefix, var_str ("n_flv"))
      call driver%write_get_int_fun (unit, prefix, var_str ("n_hel"))
      call driver%write_get_int_fun (unit, prefix, var_str ("n_col"))
      call driver%write_get_int_fun (unit, prefix, var_str ("n_cin"))
      call driver%write_get_int_fun (unit, prefix, var_str ("n_cf"))
      call driver%write_set_int_sub (unit, prefix, var_str ("flv_state"))
      call driver%write_set_int_sub (unit, prefix, var_str ("hel_state"))
      call driver%write_set_col_state_sub (unit, prefix)
      call driver%write_set_color_factors_sub (unit, prefix)
      call driver%write_get_fptr_sub (unit, prefix)
      do i = 1, driver%n_processes
         call driver%record(i)%write_wrappers (unit)
      end do
    end subroutine prclib_driver_generate_code
```

The driver module is used and required *only* if we intend to link the library
statically. Then, it provides the (static) driver type as a concrete implementa-
tion of the abstract library driver. This type contains the internal dispatcher
for assigning the library procedures to their appropriate procedure pointers. In

the dynamical case, the assignment is done via the base-type dispatcher which invokes the DL mechanism.

However, compiling this together with the rest in any case should not do any harm.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure, nopass :: write_module => prclib_driver_write_module
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prclib_driver_write_module (unit, prefix)
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Module: define library driver as an extension &
       &of the abstract driver type."
  write (unit, "(A)")  "! This is used _only_ by the library dispatcher &
       &of a static executable."
  write (unit, "(A)")  "! For a dynamical library, the stand-alone proce&
       &dures are linked via libdl."
  write (unit, "(A)")  ""
  write (unit, "(A)")  "module " &
       // char (prefix) // "driver"
  write (unit, "(A)")  "  use iso_c_binding"
  write (unit, "(A)")  "  use iso_varying_string, string_t => varying_string"
  write (unit, "(A)")  "  use diagnostics"
  write (unit, "(A)")  "  use prclib_interfaces"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "  implicit none"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "  type, extends (prclib_driver_t) :: " &
       // char (prefix) // "driver_t"
  write (unit, "(A)")  "   contains"
  write (unit, "(A)")  "     procedure :: get_c_funptr => " &
       // char (prefix) // "driver_get_c_funptr"
  write (unit, "(A)")  "  end type " &
       // char (prefix) // "driver_t"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "contains"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "  function " &
       // char (prefix) // "driver_get_c_funptr (driver, feature) result &
       &(c_fptr)"
  write (unit, "(A)")  "    class(" &
       // char (prefix) // "driver_t), intent(inout) :: driver"
  write (unit, "(A)")  "    type(string_t), intent(in) :: feature"
  write (unit, "(A)")  "    type(c_funptr) :: c_fptr"
  call write_decl ("get_n_processes", "get_n_processes")
  call write_decl ("get_stringptr", "get_process_id_ptr")
  call write_decl ("get_stringptr", "get_model_name_ptr")
  call write_decl ("get_stringptr", "get_md5sum_ptr")
  call write_decl ("get_log", "get_openmp_status")
  call write_decl ("get_int", "get_n_in")
  call write_decl ("get_int", "get_n_out")
  call write_decl ("get_int", "get_n_flv")
  call write_decl ("get_int", "get_n_hel")
```

```
      call write_decl ("get_int", "get_n_col")
      call write_decl ("get_int", "get_n_cin")
      call write_decl ("get_int", "get_n_cf")
      call write_decl ("set_int_tab1", "set_flv_state_ptr")
      call write_decl ("set_int_tab1", "set_hel_state_ptr")
      call write_decl ("set_col_state", "set_col_state_ptr")
      call write_decl ("set_color_factors", "set_color_factors_ptr")
      call write_decl ("get_fptr", "get_fptr")
      write (unit, "(A)")  "    select case (char (feature))"
      call write_case ("get_n_processes")
      call write_case ("get_process_id_ptr")
      call write_case ("get_model_name_ptr")
      call write_case ("get_md5sum_ptr")
      call write_case ("get_openmp_status")
      call write_case ("get_n_in")
      call write_case ("get_n_out")
      call write_case ("get_n_flv")
      call write_case ("get_n_hel")
      call write_case ("get_n_col")
      call write_case ("get_n_cin")
      call write_case ("get_n_cf")
      call write_case ("set_flv_state_ptr")
      call write_case ("set_hel_state_ptr")
      call write_case ("set_col_state_ptr")
      call write_case ("set_color_factors_ptr")
      call write_case ("get_fptr")
      write (unit, "(A)")  "    case default"
      write (unit, "(A)")  "       call msg_bug ('prclib2 driver setup: unknown &
           &function name')"
      write (unit, "(A)")  "    end select"
      write (unit, "(A)")  "  end function " &
           // char (prefix) // "driver_get_c_funptr"
      write (unit, "(A)")  ""
      write (unit, "(A)")  "end module " &
           // char (prefix) // "driver"
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! Stand-alone external procedures: used for both &
           &static and dynamic linkage"
    contains
      subroutine write_decl (template, feature)
        character(*), intent(in) :: template, feature
        write (unit, "(A)")  "    procedure(prc_" // template // ") &"
        write (unit, "(A)")  "         :: " &
             // char (prefix) // feature
      end subroutine write_decl
      subroutine write_case (feature)
        character(*), intent(in) :: feature
        write (unit, "(A)")  "    case ('" // feature // "')"
        write (unit, "(A)")  "       c_fptr = c_funloc (" &
             // char (prefix) // feature // ")"
      end subroutine write_case
    end subroutine prclib_driver_write_module
```

This function provides the overall library MD5sum. The function is for internal

use (therefore not bind(C)), the external interface is via the `get_md5sum_ptr`
procedure with index 0.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_lib_md5sum_fun => prclib_driver_write_lib_md5sum_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_write_lib_md5sum_fun (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! The MD5 sum of the library"
      write (unit, "(A)")  "function " // char (prefix) &
           // "md5sum () result (md5sum)"
      write (unit, "(A)")  "  implicit none"
      write (unit, "(A)")  "  character(32) :: md5sum"
      write (unit, "(A)")  "  md5sum = '" // driver%md5sum // "'"
      write (unit, "(A)")  "end function " // char (prefix) // "md5sum"
    end subroutine prclib_driver_write_lib_md5sum_fun
```

14.1.9   Interface bodies for informational functions

These interfaces implement the communication between WHIZARD (the main
program) and the process-library driver. The procedures are all BIND(C), so
they can safely be exposed by the library and handled by the `dlopen` mechanism,
which apparently understands only C calling conventions.

In the sections below, for each procedure, we provide both the interface itself
and a procedure that writes the correponding procedure as source code to the
process library driver.

Process count

Return the number of processes contained in the library.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_n_processes
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
      function prc_get_n_processes () result (n) bind(C)
        import
        integer(c_int) :: n
      end function prc_get_n_processes
    end interface
```

Here is the code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_get_n_processes_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_get_n_processes_fun (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
```

1749

```
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! Return the number of processes in this library"
      write (unit, "(A)")  "function " // char (prefix) &
           // "get_n_processes () result (n) bind(C)"
      write (unit, "(A)")  "  use iso_c_binding"
      write (unit, "(A)")  "  implicit none"
      write (unit, "(A)")  "  integer(c_int) :: n"
      write (unit, "(A,I0)")  "  n = ", driver%n_processes
      write (unit, "(A)")  "end function " // char (prefix) &
           // "get_n_processes"
    end subroutine write_get_n_processes_fun
```

Informational string functions

These functions return constant information about the matrix-element code.

The following procedures have to return strings. With the BIND(C) constraint, we choose to return the C pointer to a string, and its length, so the procedures implement this interface. They are actually subroutines.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_stringptr
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_get_stringptr (i, cptr, len) bind(C)
        import
        integer(c_int), intent(in) :: i
        type(c_ptr), intent(out) :: cptr
        integer(c_int), intent(out) :: len
      end subroutine prc_get_stringptr
    end interface
```

To hide this complication, we introduce a subroutine that converts the returned C pointer to a `string_t` object. As a side effect, we deallocate the original after conversion – otherwise, we might have a memory leak.

For the conversion, we first pointer-convert the C pointer to a Fortran character array pointer, length 1 and size `len`. Using argument association and an internal subroutine, we convert this to a character array with length `len` and size 1. Using ordinary assignment, we finally convert this to `string_t`.

The function takes the pointer-returning function as an argument. The index `i` identifies the process in the library.

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine get_string_via_cptr (string, i, get_stringptr)
      type(string_t), intent(out) :: string
      integer, intent(in) :: i
      procedure(prc_get_stringptr) :: get_stringptr
      type(c_ptr) :: cptr
      integer(c_int) :: pid, len
      character(kind=c_char), dimension(:), pointer :: c_array
      pid = i
      call get_stringptr (pid, cptr, len)
      if (c_associated (cptr)) then
         call c_f_pointer (cptr, c_array, shape = [len])
```

1750

```
        call set_string (c_array)
        call get_stringptr (0_c_int, cptr, len)
     else
        string = ""
     end if
  contains
    subroutine set_string (buffer)
      character(len, kind=c_char), dimension(1), intent(in) :: buffer
      string = buffer(1)
    end subroutine set_string
  end subroutine get_string_via_cptr
```

Since the module procedures return Fortran strings, we have to convert them.
This is the necessary auxiliary routine. The routine is not BIND(C), it is not
accessed from outside.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure, nopass :: write_string_to_array_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_string_to_array_fun (unit, prefix)
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! Auxiliary: convert character string &
          &to array pointer"
      write (unit, "(A)")  "subroutine " // char (prefix) &
          // "string_to_array (string, a)"
      write (unit, "(A)")  "  use iso_c_binding"
      write (unit, "(A)")  "  implicit none"
      write (unit, "(A)")  "  character(*), intent(in) :: string"
      write (unit, "(A)")  "  character(kind=c_char), dimension(:), &
          &allocatable, intent(out) :: a"
      write (unit, "(A)")  "  integer :: i"
      write (unit, "(A)")  "  allocate (a (len (string)))"
      write (unit, "(A)")  "  do i = 1, size (a)"
      write (unit, "(A)")  "     a(i) = string(i:i)"
      write (unit, "(A)")  "  end do"
      write (unit, "(A)")  "end subroutine " // char (prefix) &
          // "string_to_array"
    end subroutine write_string_to_array_fun
```

The above routine is called by other functions. It is not in a module, so they
need its interface explicitly.

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_string_to_array_interface (unit, prefix)
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      write (unit, "(2x,A)")  "interface"
      write (unit, "(2x,A)")  "   subroutine " // char (prefix) &
          // "string_to_array (string, a)"
      write (unit, "(2x,A)")  "     use iso_c_binding"
      write (unit, "(2x,A)")  "     implicit none"
      write (unit, "(2x,A)")  "     character(*), intent(in) :: string"
```

```
        write (unit, "(2x,A)")  "    character(kind=c_char), dimension(:), &
            &allocatable, intent(out) :: a"
        write (unit, "(2x,A)")  "   end subroutine " // char (prefix) &
            // "string_to_array"
        write (unit, "(2x,A)")  "end interface"
    end subroutine write_string_to_array_interface
```

Here are the info functions which return strings, implementing the interface
`prc_get_stringptr`.

Return the process ID for each process.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_get_process_id_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_get_process_id_fun (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      integer :: i
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! Return the process ID of process #i &
          &(as a C pointer to a character array)"
      write (unit, "(A)")  "subroutine " // char (prefix) &
          // "get_process_id_ptr (i, cptr, len) bind(C)"
      write (unit, "(A)")  "  use iso_c_binding"
      write (unit, "(A)")  "  implicit none"
      write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
      write (unit, "(A)")  "  type(c_ptr), intent(out) :: cptr"
      write (unit, "(A)")  "  integer(c_int), intent(out) :: len"
      write (unit, "(A)")  "  character(kind=c_char), dimension(:), &
          &allocatable, target, save :: a"
      call write_string_to_array_interface (unit, prefix)
      write (unit, "(A)")  "  select case (i)"
      write (unit, "(A)")  "  case (0);  if (allocated (a))  deallocate (a)"
      do i = 1, driver%n_processes
         write (unit, "(A,I0,9A)")  "  case (", i, ");  ", &
             "call ", char (prefix), "string_to_array ('", &
             char (driver%record(i)%id), "', a)"
      end do
      write (unit, "(A)")  "  end select"
      write (unit, "(A)")  "  if (allocated (a)) then"
      write (unit, "(A)")  "     cptr = c_loc (a)"
      write (unit, "(A)")  "     len = size (a)"
      write (unit, "(A)")  "  else"
      write (unit, "(A)")  "     cptr = c_null_ptr"
      write (unit, "(A)")  "     len = 0"
      write (unit, "(A)")  "  end if"
      write (unit, "(A)")  "end subroutine " // char (prefix) &
          // "get_process_id_ptr"
    end subroutine write_get_process_id_fun
```

Return the model name, given explicitly.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_get_model_name_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_get_model_name_fun (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      integer :: i
      write (unit, "(A)")   ""
      write (unit, "(A)")   "! Return the model name for process #i &
          &(as a C pointer to a character array)"
      write (unit, "(A)")   "subroutine " // char (prefix) &
          // "get_model_name_ptr (i, cptr, len) bind(C)"
      write (unit, "(A)")   "  use iso_c_binding"
      write (unit, "(A)")   "  implicit none"
      write (unit, "(A)")   "  integer(c_int), intent(in) :: i"
      write (unit, "(A)")   "  type(c_ptr), intent(out) :: cptr"
      write (unit, "(A)")   "  integer(c_int), intent(out) :: len"
      write (unit, "(A)")   "  character(kind=c_char), dimension(:), &
          &allocatable, target, save :: a"
      call write_string_to_array_interface (unit, prefix)
      write (unit, "(A)")   "  select case (i)"
      write (unit, "(A)")   "  case (0);  if (allocated (a))  deallocate (a)"
      do i = 1, driver%n_processes
         write (unit, "(A,I0,9A)")  "  case (", i, ");  ", &
              "call ", char (prefix), "string_to_array ('" , &
              char (driver%record(i)%model_name), &
              "', a)"
      end do
      write (unit, "(A)")   "  end select"
      write (unit, "(A)")   "  if (allocated (a)) then"
      write (unit, "(A)")   "     cptr = c_loc (a)"
      write (unit, "(A)")   "     len = size (a)"
      write (unit, "(A)")   "  else"
      write (unit, "(A)")   "     cptr = c_null_ptr"
      write (unit, "(A)")   "     len = 0"
      write (unit, "(A)")   "  end if"
      write (unit, "(A)")   "end subroutine " // char (prefix) &
          // "get_model_name_ptr"
    end subroutine write_get_model_name_fun
```

Call the MD5 sum function for the process. The function calls the corresponding
function of the matrix-element code, and it returns the C address of a character
array with length 32.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_get_md5sum_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_get_md5sum_fun (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      integer :: i
      write (unit, "(A)")   ""
      write (unit, "(A)")   "! Return the MD5 sum for the process configuration &
          &(as a C pointer to a character array)"
```

```
      write (unit, "(A)")  "subroutine " // char (prefix) &
           // "get_md5sum_ptr (i, cptr, len) bind(C)"
      write (unit, "(A)")  "  use iso_c_binding"
      call driver%write_interfaces (unit, var_str ("md5sum"))
      write (unit, "(A)")  "  interface"
      write (unit, "(A)")  "     function " // char (prefix) &
           // "md5sum () result (md5sum)"
      write (unit, "(A)")  "       character(32) :: md5sum"
      write (unit, "(A)")  "     end function " // char (prefix) // "md5sum"
      write (unit, "(A)")  "  end interface"
      write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
      write (unit, "(A)")  "  type(c_ptr), intent(out) :: cptr"
      write (unit, "(A)")  "  integer(c_int), intent(out) :: len"
      write (unit, "(A)")  "  character(kind=c_char), dimension(32), &
           &target, save :: md5sum"
      write (unit, "(A)")  "  select case (i)"
      write (unit, "(A)")  "  case (0)"
      write (unit, "(A)")  "     call copy (" // char (prefix) // "md5sum ())"
      write (unit, "(A)")  "     cptr = c_loc (md5sum)"
      do i = 1, driver%n_processes
         write (unit, "(A,I0,A)")  "  case (", i, ")"
         call driver%record(i)%write_md5sum_call (unit)
      end do
      write (unit, "(A)")  "  case default"
      write (unit, "(A)")  "     cptr = c_null_ptr"
      write (unit, "(A)")  "  end select"
      write (unit, "(A)")  "  len = 32"
      write (unit, "(A)")  "contains"
      write (unit, "(A)")  "  subroutine copy (md5sum_tmp)"
      write (unit, "(A)")  "    character, dimension(32), intent(in) :: &
           &md5sum_tmp"
      write (unit, "(A)")  "    md5sum = md5sum_tmp"
      write (unit, "(A)")  "  end subroutine copy"
      write (unit, "(A)")  "end subroutine " // char (prefix) &
             // "get_md5sum_ptr"
    end subroutine write_get_md5sum_fun
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
    procedure :: write_md5sum_call => prclib_driver_record_write_md5sum_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_record_write_md5sum_call (record, unit)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      call record%writer%write_md5sum_call (unit, record%id)
    end subroutine prclib_driver_record_write_md5sum_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
    procedure(write_code), deferred :: write_md5sum_call
```

In the Fortran module case, we take a detour. The string returned by the
Fortran function is copied into a fixed-size array. The copy routine is an internal

1754

subroutine of `get_md5sum_ptr`. We return the C address of the target array.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
procedure :: write_md5sum_call => prc_writer_f_module_write_md5sum_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_f_module_write_md5sum_call (writer, unit, id)
  class(prc_writer_f_module_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "call copy (", &
       char (writer%get_c_procname (id, var_str ("md5sum"))), " ())"
  write (unit, "(5x,9A)")  "cptr = c_loc (md5sum)"
end subroutine prc_writer_f_module_write_md5sum_call
```

In the C library case, the library function returns a C pointer, which we can just copy.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩≡
```
procedure :: write_md5sum_call => prc_writer_c_lib_write_md5sum_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_c_lib_write_md5sum_call (writer, unit, id)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)") &
       "cptr =  ", &
       char (writer%get_c_procname (id, var_str ("get_md5sum"))), " ()"
end subroutine prc_writer_c_lib_write_md5sum_call
```

Actual references to the info functions

The string-valued info functions return C character arrays. For the API of the library driver, we provide convenience functions which (re)convert those arrays into `string_t` objects.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: get_process_id => prclib_driver_get_process_id
procedure :: get_model_name => prclib_driver_get_model_name
procedure :: get_md5sum => prclib_driver_get_md5sum
```

⟨*Prclib interfaces: procedures*⟩+≡
```
function prclib_driver_get_process_id (driver, i) result (string)
  type(string_t) :: string
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  call get_string_via_cptr (string, i, driver%get_process_id_ptr)
end function prclib_driver_get_process_id

function prclib_driver_get_model_name (driver, i) result (string)
  type(string_t) :: string
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  call get_string_via_cptr (string, i, driver%get_model_name_ptr)
end function prclib_driver_get_model_name
```

1755

```
function prclib_driver_get_md5sum (driver, i) result (string)
  type(string_t) :: string
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  call get_string_via_cptr (string, i, driver%get_md5sum_ptr)
end function prclib_driver_get_md5sum
```

Informational logical functions

When returning a logical value, we use the C boolean type, which may differ
from Fortran.

⟨*Prclib interfaces: public*⟩+≡
```
public :: prc_get_log
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
abstract interface
   function prc_get_log (pid) result (l) bind(C)
     import
     integer(c_int), intent(in) :: pid
     logical(c_bool) :: l
   end function prc_get_log
end interface
```

Return a logical flag which tells whether OpenMP is supported for a specific
process code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: write_get_openmp_status_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine write_get_openmp_status_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Return the OpenMP support status"
  write (unit, "(A)")  "function " // char (prefix) &
       // "get_openmp_status (i) result (openmp_status) bind(C)"
  write (unit, "(A)")  "  use iso_c_binding"
  call driver%write_interfaces (unit, var_str ("openmp_supported"))
  write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
  write (unit, "(A)")  "  logical(c_bool) :: openmp_status"
  write (unit, "(A)")  "  select case (i)"
  do i = 1, driver%n_processes
     write (unit, "(A,I0,9A)")  "  case (", i, ");  ", &
            "openmp_status = ", &
            char (driver%record(i)%get_c_procname &
            (var_str ("openmp_supported"))), " ()"
  end do
  write (unit, "(A)")  "  end select"
  write (unit, "(A)")  "end function " // char (prefix) &
       // "get_openmp_status"
end subroutine write_get_openmp_status_fun
```

Informational integer functions

Various process metadata are integer values. We can use a single interface for all of them.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_int
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       function prc_get_int (pid) result (n) bind(C)
         import
         integer(c_int), intent(in) :: pid
         integer(c_int) :: n
       end function prc_get_int
    end interface
```

This function returns any data of type integer, for each process.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_get_int_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_get_int_fun (driver, unit, prefix, feature)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      type(string_t), intent(in) :: feature
      integer :: i
      write (unit, "(A)")  ""
      write (unit, "(9A)")  "! Return the value of ", char (feature)
      write (unit, "(9A)")  "function ", char (prefix), &
           "get_", char (feature), " (pid)", &
           " result (", char (feature), ") bind(C)"
      write (unit, "(9A)")  "  use iso_c_binding"
      call driver%write_interfaces (unit, feature)
      write (unit, "(9A)")  "  integer(c_int), intent(in) :: pid"
      write (unit, "(9A)")  "  integer(c_int) :: ", char (feature)
      write (unit, "(9A)")  "  select case (pid)"
      do i = 1, driver%n_processes
         write (unit, "(2x,A,I0,9A)")  "case (", i, ");  ", &
              char (feature), " = ", &
              char (driver%record(i)%get_c_procname (feature)), &
              " ()"
      end do
      write (unit, "(9A)")  "  end select"
      write (unit, "(9A)")  "end function ", char (prefix), &
           "get_", char (feature)
    end subroutine write_get_int_fun
```

Write a `case` line that assigns the value of the external function to the current return value.

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_case_int_fun (record, unit, i, feature)
```

```
          class(prclib_driver_record_t), intent(in) :: record
          integer, intent(in) :: unit
          integer, intent(in) :: i
          type(string_t), intent(in) :: feature
          write (unit, "(5x,A,I0,9A)")  "case (", i, ");  ", &
               char (feature), " = ", char (record%get_c_procname (feature))
        end subroutine write_case_int_fun
```

Flavor and helicity tables

Transferring tables is more complicated. First, a two-dimensional array.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_set_int_tab1
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_set_int_tab1 (pid, tab, shape) bind(C)
         import
         integer(c_int), intent(in) :: pid
         integer(c_int), dimension(*), intent(out) :: tab
         integer(c_int), dimension(2), intent(in) :: shape
       end subroutine prc_set_int_tab1
    end interface
```

This subroutine returns a table of integers.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_set_int_sub
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_set_int_sub (driver, unit, prefix, feature)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      type(string_t), intent(in) :: feature
      integer :: i
      write (unit, "(A)")  ""
      write (unit, "(9A)")  "! Set table: ", char (feature)
      write (unit, "(9A)")  "subroutine ", char (prefix), &
           "set_", char (feature), "_ptr (pid, ", char (feature), &
           ", shape) bind(C)"
      write (unit, "(9A)")  "  use iso_c_binding"
      call driver%write_interfaces (unit, feature)
      write (unit, "(9A)")  "  integer(c_int), intent(in) :: pid"
      write (unit, "(9A)")  "  integer(c_int), dimension(*), intent(out) :: ", &
           char (feature)
      write (unit, "(9A)")  "  integer(c_int), dimension(2), intent(in) :: shape"
      write (unit, "(9A)")  "  integer, dimension(:,:), allocatable :: ", &
           char (feature), "_tmp"
      write (unit, "(9A)")  "  integer :: i, j"
      write (unit, "(9A)")  "  select case (pid)"
      do i = 1, driver%n_processes
         write (unit, "(2x,A,I0,A)")  "case (", i, ")"
         call driver%record(i)%write_int_sub_call (unit, feature)
      end do
```

1758

```
      write (unit, "(9A)")  "  end select"
      write (unit, "(9A)")  "end subroutine ", char (prefix), &
           "set_", char (feature), "_ptr"
    end subroutine write_set_int_sub
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
    procedure :: write_int_sub_call => prclib_driver_record_write_int_sub_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_record_write_int_sub_call (record, unit, feature)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      type(string_t), intent(in) :: feature
      call record%writer%write_int_sub_call (unit, record%id, feature)
    end subroutine prclib_driver_record_write_int_sub_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
    procedure(write_feature_code), deferred :: write_int_sub_call
```

In the Fortran module case, we need an extra copy in the (academical) situation
where default integer and `c_int` differ. Otherwise, we just associate a Fortran
array with the C pointer and let the matrix-element subroutine fill the array.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
    procedure :: write_int_sub_call => prc_writer_f_module_write_int_sub_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prc_writer_f_module_write_int_sub_call (writer, unit, id, feature)
      class(prc_writer_f_module_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, "(5x,9A)")  "allocate (",  char (feature), "_tmp ", &
           "(shape(1), shape(2)))"
      write (unit, "(5x,9A)")  "call ", &
           char (writer%get_c_procname (id, feature)), &
           " (", char (feature), "_tmp)"
      write (unit, "(5x,9A)")  "forall (i=1:shape(1), j=1:shape(2)) "
      write (unit, "(8x,9A)")  char (feature), "(i + shape(1)*(j-1)) = ", &
           char (feature), "_tmp", "(i,j)"
      write (unit, "(5x,9A)")  "end forall"
    end subroutine prc_writer_f_module_write_int_sub_call
```

In the C library case, we just transfer the C pointer to the library function.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡
```
    procedure :: write_int_sub_call => prc_writer_c_lib_write_int_sub_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prc_writer_c_lib_write_int_sub_call (writer, unit, id, feature)
      class(prc_writer_c_lib_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, "(5x,9A)")  "call ", &
           char (writer%get_c_procname (id, feature)), " (", char (feature), ")"
```

```
      end subroutine prc_writer_c_lib_write_int_sub_call
```

Color state table

The color-state specification needs a table of integers (one array per color flow)
and a corresponding array of color-ghost flags.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_set_col_state
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_set_col_state (pid, col_state, ghost_flag, shape) bind(C)
         import
         integer(c_int), intent(in) :: pid
         integer(c_int), dimension(*), intent(out) :: col_state
         logical(c_bool), dimension(*), intent(out) :: ghost_flag
         integer(c_int), dimension(3), intent(in) :: shape
       end subroutine prc_set_col_state
    end interface
```

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_set_col_state_sub
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_set_col_state_sub (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      integer :: i
      type(string_t) :: feature
      feature = "col_state"
      write (unit, "(A)")  ""
      write (unit, "(9A)")  "! Set tables: col_state, ghost_flag"
      write (unit, "(9A)")  "subroutine ", char (prefix), &
           "set_col_state_ptr (pid, col_state, ghost_flag, shape) bind(C)"
      write (unit, "(9A)")  "  use iso_c_binding"
      call driver%write_interfaces (unit, feature)
      write (unit, "(9A)")  "  integer(c_int), intent(in) :: pid"
      write (unit, "(9A)") &
           "  integer(c_int), dimension(*), intent(out) :: col_state"
      write (unit, "(9A)") &
           "  logical(c_bool), dimension(*), intent(out) :: ghost_flag"
      write (unit, "(9A)") &
           "  integer(c_int), dimension(3), intent(in) :: shape"
      write (unit, "(9A)") &
           "  integer, dimension(:,:,:), allocatable :: col_state_tmp"
      write (unit, "(9A)") &
           "  logical, dimension(:,:), allocatable :: ghost_flag_tmp"
      write (unit, "(9A)") "  integer :: i, j, k"
      write (unit, "(A)")  "  select case (pid)"
      do i = 1, driver%n_processes
         write (unit, "(A,I0,A)")  "  case (", i, ")"
         call driver%record(i)%write_col_state_call (unit)
      end do
```

```
          write (unit, "(A)")  "  end select"
          write (unit, "(9A)")  "end subroutine ", char (prefix), &
               "set_col_state_ptr"
        end subroutine write_set_col_state_sub
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
        procedure :: write_col_state_call => prclib_driver_record_write_col_state_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
        subroutine prclib_driver_record_write_col_state_call (record, unit)
          class(prclib_driver_record_t), intent(in) :: record
          integer, intent(in) :: unit
          call record%writer%write_col_state_call (unit, record%id)
        end subroutine prclib_driver_record_write_col_state_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
        procedure(write_code), deferred :: write_col_state_call
```

In the Fortran module case, we need an extra copy in the (academical) situation
where default integer and `c_int` differ. Otherwise, we just associate a Fortran
array with the C pointer and let the matrix-element subroutine fill the array.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
        procedure :: write_col_state_call => prc_writer_f_module_write_col_state_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
        subroutine prc_writer_f_module_write_col_state_call (writer, unit, id)
          class(prc_writer_f_module_t), intent(in) :: writer
          integer, intent(in) :: unit
          type(string_t), intent(in) :: id
          write (unit, "(9A)")  "  allocate (col_state_tmp ", &
                  "(shape(1), shape(2), shape(3)))"
          write (unit, "(5x,9A)")  "allocate (ghost_flag_tmp ", &
                    "(shape(2), shape(3)))"
          write (unit, "(5x,9A)")  "call ", &
              char (writer%get_c_procname (id, var_str ("col_state"))), &
              " (col_state_tmp, ghost_flag_tmp)"
          write (unit, "(5x,9A)")  "forall (i = 1:shape(2), j = 1:shape(3))"
          write (unit, "(8x,9A)")  "forall (k = 1:shape(1))"
          write (unit, "(11x,9A)")  &
              "col_state(k + shape(1) * (i + shape(2)*(j-1) - 1)) ", &
              "= col_state_tmp(k,i,j)"
          write (unit, "(8x,9A)")  "end forall"
          write (unit, "(8x,9A)")  &
              "ghost_flag(i + shape(2)*(j-1)) = ghost_flag_tmp(i,j)"
          write (unit, "(5x,9A)")  "end forall"
        end subroutine prc_writer_f_module_write_col_state_call
```

In the C library case, we just transfer the C pointer to the library function.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡
```
        procedure :: write_col_state_call => prc_writer_c_lib_write_col_state_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prc_writer_c_lib_write_col_state_call (writer, unit, id)
      class(prc_writer_c_lib_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      write (unit, "(5x,9A)")  "call ", &
           char (writer%get_c_procname (id, var_str ("col_state"))), &
           " (col_state, ghost_flag)"
    end subroutine prc_writer_c_lib_write_col_state_call
```

Color factors

For the color-factor information, we return two integer arrays and a complex
array.

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_set_color_factors
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_set_color_factors &
            (pid, cf_index1, cf_index2, color_factors, shape) bind(C)
         import
         integer(c_int), intent(in) :: pid
         integer(c_int), dimension(*), intent(out) :: cf_index1, cf_index2
         complex(c_default_complex), dimension(*), intent(out) :: color_factors
         integer(c_int), dimension(1), intent(in) :: shape
       end subroutine prc_set_color_factors
    end interface
```

This subroutine returns the color-flavor factor table.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_set_color_factors_sub
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_set_color_factors_sub (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      integer :: i
      type(string_t) :: feature
      feature = "color_factors"
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! Set tables: color factors"
      write (unit, "(9A)")  "subroutine ", char (prefix), &
           "set_color_factors_ptr (pid, cf_index1, cf_index2, color_factors, ", &
           "shape) bind(C)"
      write (unit, "(A)")  "  use iso_c_binding"
      write (unit, "(A)")  "  use kinds"
      write (unit, "(A)")  "  use omega_color"
      call driver%write_interfaces (unit, feature)
      write (unit, "(A)")  "  integer(c_int), intent(in) :: pid"
      write (unit, "(A)")  "  integer(c_int), dimension(1), intent(in) :: shape"
      write (unit, "(A)")  "  integer(c_int), dimension(*), intent(out) :: &
```

```
              &cf_index1, cf_index2"
      write (unit, "(A)")  "  complex(c_default_complex), dimension(*), &
              &intent(out) :: color_factors"
      write (unit, "(A)")  "  type(omega_color_factor), dimension(:), &
              &allocatable :: cf"
      write (unit, "(A)")  "  select case (pid)"
      do i = 1, driver%n_processes
          write (unit, "(2x,A,I0,A)")  "case (", i, ")"
          call driver%record(i)%write_color_factors_call (unit)
      end do
      write (unit, "(A)")  "  end select"
      write (unit, "(A)")  "end subroutine " // char (prefix) &
              // "set_color_factors_ptr"
  end subroutine write_set_color_factors_sub
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡
```
    procedure :: write_color_factors_call => prclib_driver_record_write_color_factors_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_record_write_color_factors_call (record, unit)
      class(prclib_driver_record_t), intent(in) :: record
      integer, intent(in) :: unit
      call record%writer%write_color_factors_call (unit, record%id)
    end subroutine prclib_driver_record_write_color_factors_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
    procedure(write_code), deferred :: write_color_factors_call
```

In the Fortran module case, the matrix-element procedure fills an array of
`omega_color_factor` elements. We distribute this array among two integer
arrays and one complex-valued array, for which we have the C pointers.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
    procedure :: write_color_factors_call => prc_writer_f_module_write_color_factors_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prc_writer_f_module_write_color_factors_call (writer, unit, id)
      class(prc_writer_f_module_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      write (unit, "(5x,A)")  "allocate (cf (shape(1)))"
      write (unit, "(5x,9A)")  "call ", &
          char (writer%get_c_procname (id, var_str ("color_factors"))), " (cf)"
      write (unit, "(5x,9A)")  "cf_index1(1:shape(1)) = cf%i1"
      write (unit, "(5x,9A)")  "cf_index2(1:shape(1)) = cf%i2"
      write (unit, "(5x,9A)")  "color_factors(1:shape(1)) = cf%factor"
    end subroutine prc_writer_f_module_write_color_factors_call
```

In the C library case, we just transfer the C pointers to the library function.
There are three arrays.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡
```
    procedure :: write_color_factors_call => &
        prc_writer_c_lib_write_color_factors_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prc_writer_c_lib_write_color_factors_call (writer, unit, id)
      class(prc_writer_c_lib_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      write (unit, "(5x,9A)")  "call ", &
            char (writer%get_c_procname (id, var_str ("color_factors"))), &
            " (cf_index1, cf_index2, color_factors)"
    end subroutine prc_writer_c_lib_write_color_factors_call
```

14.1.10   Interfaces for C-library matrix element

If the matrix element code is not provided as a Fortran module but as a C or
bind(C) Fortran library, we need explicit interfaces for the library functions.
They are not identical to the Fortran module versions. They transfer pointers
directly.

The implementation is part of the `prc_writer_c_lib` type, which serves as
base type for all C-library writers. It writes specific interfaces depending on the
feature.

We bind this as the method `write_standard_interface` instead of `write_interface`,
because we have to override the latter. Otherwise we could not call the method
because the writer type is abstract.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡
```
    procedure :: write_standard_interface => prc_writer_c_lib_write_interface
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prc_writer_c_lib_write_interface (writer, unit, id, feature)
      class(prc_writer_c_lib_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      select case (char (feature))
      case ("md5sum")
         write (unit, "(2x,9A)")  "interface"
         write (unit, "(5x,9A)")  "function ", &
               char (writer%get_c_procname (id, var_str ("get_md5sum"))), &
               " () result (cptr) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "implicit none"
         write (unit, "(7x,9A)")  "type(c_ptr) :: cptr"
         write (unit, "(5x,9A)")  "end function ", &
               char (writer%get_c_procname (id, var_str ("get_md5sum")))
         write (unit, "(2x,9A)")  "end interface"
      case ("openmp_supported")
         write (unit, "(2x,9A)")  "interface"
         write (unit, "(5x,9A)")  "function ", &
               char (writer%get_c_procname (id, feature)), &
               " () result (status) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "implicit none"
         write (unit, "(7x,9A)")  "logical(c_bool) :: status"
         write (unit, "(5x,9A)")  "end function ", &
               char (writer%get_c_procname (id, feature))
```

```
                  write (unit, "(2x,9A)")  "end interface"
          case ("n_in", "n_out", "n_flv", "n_hel", "n_col", "n_cin", "n_cf")
                  write (unit, "(2x,9A)")  "interface"
                  write (unit, "(5x,9A)")  "function ", &
                       char (writer%get_c_procname (id, feature)), &
                       " () result (n) bind(C)"
                  write (unit, "(7x,9A)")  "import"
                  write (unit, "(7x,9A)")  "implicit none"
                  write (unit, "(7x,9A)")  "integer(c_int) :: n"
                  write (unit, "(5x,9A)")  "end function ", &
                       char (writer%get_c_procname (id, feature))
                  write (unit, "(2x,9A)")  "end interface"
          case ("flv_state", "hel_state")
                  write (unit, "(2x,9A)")  "interface"
                  write (unit, "(5x,9A)")  "subroutine ", &
                       char (writer%get_c_procname (id, feature)), &
                       " (", char (feature), ") bind(C)"
                  write (unit, "(7x,9A)")  "import"
                  write (unit, "(7x,9A)")  "implicit none"
                  write (unit, "(7x,9A)")  "integer(c_int), dimension(*), intent(out) ", &
                       ":: ", char (feature)
                  write (unit, "(5x,9A)")  "end subroutine ", &
                       char (writer%get_c_procname (id, feature))
                  write (unit, "(2x,9A)")  "end interface"
          case ("col_state")
                  write (unit, "(2x,9A)")  "interface"
                  write (unit, "(5x,9A)")  "subroutine ", &
                       char (writer%get_c_procname (id, feature)), &
                       " (col_state, ghost_flag) bind(C)"
                  write (unit, "(7x,9A)")  "import"
                  write (unit, "(7x,9A)")  "implicit none"
                  write (unit, "(7x,9A)")  "integer(c_int), dimension(*), intent(out) ", &
                       ":: col_state"
                  write (unit, "(7x,9A)")  "logical(c_bool), dimension(*), intent(out) ", &
                       ":: ghost_flag"
                  write (unit, "(5x,9A)")  "end subroutine ", &
                       char (writer%get_c_procname (id, feature))
                  write (unit, "(2x,9A)")  "end interface"
          case ("color_factors")
                  write (unit, "(2x,9A)")  "interface"
                  write (unit, "(5x,9A)")  "subroutine ", &
                       char (writer%get_c_procname (id, feature)), &
                       " (cf_index1, cf_index2, color_factors) bind(C)"
                  write (unit, "(7x,9A)")  "import"
                  write (unit, "(7x,9A)")  "implicit none"
                  write (unit, "(7x,9A)")  "integer(c_int), dimension(*), &
                       &intent(out) :: cf_index1"
                  write (unit, "(7x,9A)")  "integer(c_int), dimension(*), &
                       &intent(out) :: cf_index2"
                  write (unit, "(7x,9A)")  "complex(c_default_complex), dimension(*), &
                       &intent(out) :: color_factors"
                  write (unit, "(5x,9A)")  "end subroutine ", &
                       char (writer%get_c_procname (id, feature))
                  write (unit, "(2x,9A)")  "end interface"
```

```
      end select
    end subroutine prc_writer_c_lib_write_interface
```

14.1.11 Retrieving the tables

In the previous section we had the writer routines for procedures that return
tables, actually C pointers to tables. Here, we write convenience routines that
unpack them and move the contents to suitable Fortran arrays.

The flavor and helicity tables are two-dimensional integer arrays. We use
intermediate storage for correctly transforming C to Fortran data types.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: set_flv_state => prclib_driver_set_flv_state
    procedure :: set_hel_state => prclib_driver_set_hel_state
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_set_flv_state (driver, i, flv_state)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: i
      integer, dimension(:,:), allocatable, intent(out) :: flv_state
      integer :: n_tot, n_flv
      integer(c_int) :: pid
      integer(c_int), dimension(:,:), allocatable :: c_flv_state
      pid = i
      n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
      n_flv = driver%get_n_flv (pid)
      allocate (flv_state (n_tot, n_flv))
      allocate (c_flv_state (n_tot, n_flv))
      call driver%set_flv_state_ptr &
           (pid, c_flv_state, int ([n_tot, n_flv], kind=c_int))
      flv_state = c_flv_state
    end subroutine prclib_driver_set_flv_state

    subroutine prclib_driver_set_hel_state (driver, i, hel_state)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: i
      integer, dimension(:,:), allocatable, intent(out) :: hel_state
      integer :: n_tot, n_hel
      integer(c_int) :: pid
      integer(c_int), dimension(:,:), allocatable, target :: c_hel_state
      pid = i
      n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
      n_hel = driver%get_n_hel (pid)
      allocate (hel_state (n_tot, n_hel))
      allocate (c_hel_state (n_tot, n_hel))
      call driver%set_hel_state_ptr &
           (pid, c_hel_state, int ([n_tot, n_hel], kind=c_int))
      hel_state = c_hel_state
    end subroutine prclib_driver_set_hel_state
```

The color-flow table is three-dimensional, otherwise similar. We simultaneously
set the ghost-flag table, which consists of logical entries.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
      procedure :: set_col_state => prclib_driver_set_col_state
⟨Prclib interfaces: procedures⟩+≡
    subroutine prclib_driver_set_col_state (driver, i, col_state, ghost_flag)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: i
      integer, dimension(:,:,:), allocatable, intent(out) :: col_state
      logical, dimension(:,:), allocatable, intent(out) :: ghost_flag
      integer :: n_cin, n_tot, n_col
      integer(c_int) :: pid
      integer(c_int), dimension(:,:,:), allocatable :: c_col_state
      logical(c_bool), dimension(:,:), allocatable :: c_ghost_flag
      pid = i
      n_cin = driver%get_n_cin (pid)
      n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
      n_col = driver%get_n_col (pid)
      allocate (col_state (n_cin, n_tot, n_col))
      allocate (c_col_state (n_cin, n_tot, n_col))
      allocate (ghost_flag (n_tot, n_col))
      allocate (c_ghost_flag (n_tot, n_col))
      call driver%set_col_state_ptr (pid, &
           c_col_state, c_ghost_flag, int ([n_cin, n_tot, n_col], kind=c_int))
      col_state = c_col_state
      ghost_flag = c_ghost_flag
    end subroutine prclib_driver_set_col_state
```

The color-factor table is a sparse matrix: a two-column array of indices and one
array which contains the corresponding factors.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: set_color_factors => prclib_driver_set_color_factors
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_set_color_factors (driver, i, color_factors, cf_index)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: i
      complex(default), dimension(:), allocatable, intent(out) :: color_factors
      integer, dimension(:,:), allocatable, intent(out) :: cf_index
      integer :: n_cf
      integer(c_int) :: pid
      complex(c_default_complex), dimension(:), allocatable, target :: c_color_factors
      integer(c_int), dimension(:), allocatable, target :: c_cf_index1
      integer(c_int), dimension(:), allocatable, target :: c_cf_index2
      pid = i
      n_cf = driver%get_n_cf (pid)
      allocate (color_factors (n_cf))
      allocate (c_color_factors (n_cf))
      allocate (c_cf_index1 (n_cf))
      allocate (c_cf_index2 (n_cf))
      call driver%set_color_factors_ptr (pid, &
           c_cf_index1, c_cf_index2, &
           c_color_factors, int ([n_cf], kind=c_int))
      color_factors = c_color_factors
      allocate (cf_index (2, n_cf))
      cf_index(1,:) = c_cf_index1
      cf_index(2,:) = c_cf_index2
```

```
            end subroutine prclib_driver_set_color_factors
```

14.1.12 Returning a procedure pointer

The functions that directly access the matrix element, event by event, are assigned to a process-specific driver object as procedure pointers. For the `dlopen` interface, we use C function pointers. This subroutine returns such a pointer:

⟨*Prclib interfaces: public*⟩+≡
```
    public :: prc_get_fptr
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_get_fptr (pid, fid, fptr) bind(C)
         import
         integer(c_int), intent(in) :: pid
         integer(c_int), intent(in) :: fid
         type(c_funptr), intent(out) :: fptr
       end subroutine prc_get_fptr
    end interface
```

This procedure writes the source code for the procedure pointer returning subroutine.

All C functions that are provided by the matrix element code of a specific process are handled here. The selection consists of a double layered `select case` construct.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: write_get_fptr_sub
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine write_get_fptr_sub (driver, unit, prefix)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: unit
      type(string_t), intent(in) :: prefix
      integer :: i, j
      write (unit, "(A)")  ""
      write (unit, "(A)")  "! Return C pointer to a procedure:"
      write (unit, "(A)")  "! pid = process index;  fid = function index"
      write (unit, "(4A)")  "subroutine ", char (prefix), "get_fptr ", &
           "(pid, fid, fptr) bind(C)"
      write (unit, "(A)")  "  use iso_c_binding"
      write (unit, "(A)")  "  use kinds"
      write (unit, "(A)")  "  implicit none"
      write (unit, "(A)")  "  integer(c_int), intent(in) :: pid"
      write (unit, "(A)")  "  integer(c_int), intent(in) :: fid"
      write (unit, "(A)")  "  type(c_funptr), intent(out) :: fptr"
      do i = 1, driver%n_processes
         call driver%record(i)%write_interfaces (unit)
      end do
      write (unit, "(A)")  "  select case (pid)"
      do i = 1, driver%n_processes
         write (unit, "(2x,A,I0,A)")  "case (", i, ")"
         write (unit, "(5x,A)")  "select case (fid)"
```

```
         associate (record => driver%record(i))
           do j = 1, size (record%feature)
             write (unit, "(5x,A,I0,9A)")  "case (", j, ");  ", &
                   "fptr = c_funloc (", &
                   char (record%get_c_procname (record%feature(j))), &
                   ")"
           end do
         end associate
         write (unit, "(5x,A)")  "end select"
       end do
       write (unit, "(A)")  "  end select"
       write (unit, "(3A)")  "end subroutine ", char (prefix), "get_fptr"
     end subroutine write_get_fptr_sub
```

The procedures for which we want to return a pointer (the 'features' of the
matrix element code) are actually Fortran module procedures. If we want to
have a C signature, we must write wrapper functions for all of them. The
procedures, their signatures, and the appropriate writer routines are specific for
the process type.

To keep this generic, we do not provide the writer routines here, but just
the interface for a writer routine. The actual routines are stored in the process
record.

The prefix indicates the library, the id indicates the process, and procname
is the bare name of the procedure to be written.

⟨*Prclib interfaces: public*⟩+≡
```
     public :: write_driver_code
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
     abstract interface
       subroutine write_driver_code (unit, prefix, id, procname)
         import
         integer, intent(in) :: unit
         type(string_t), intent(in) :: prefix
         type(string_t), intent(in) :: id
         type(string_t), intent(in) :: procname
       end subroutine write_driver_code
     end interface
```

14.1.13   Hooks

Interface for additional library unload / reload hooks (currently unused!)

⟨*Prclib interfaces: public*⟩+≡
```
     public :: prclib_unload_hook
     public :: prclib_reload_hook
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
     abstract interface
       subroutine prclib_unload_hook (libname)
         import
         type(string_t), intent(in) :: libname
       end subroutine prclib_unload_hook
```

```
      subroutine prclib_reload_hook (libname)
        import
        type(string_t), intent(in) :: libname
      end subroutine prclib_reload_hook
    end interface
```

14.1.14 Make source, compile, link

Since we should have written a Makefile, these tasks amount to simple `make`
calls. Note that the Makefile targets depend on each other, so calling `link`
executes also the `source` and `compile` steps, when necessary.

The first routine writes source-code files for the individual processes. First
it calls the writer routines directly for each process, then it calls `make source`.
The make command may either post-process the files, or it may do the complete
work, e.g., calling an external program the generates the files.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: make_source => prclib_driver_make_source
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_make_source (driver, os_data)
      class(prclib_driver_t), intent(in) :: driver
      type(os_data_t), intent(in) :: os_data
      integer :: i
      do i = 1, driver%n_processes
        call driver%record(i)%write_source_code ()
      end do
      call os_system_call ("make source " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
    end subroutine prclib_driver_make_source
```

Compile matrix element source code and the driver source code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: make_compile => prclib_driver_make_compile
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_make_compile (driver, os_data)
      class(prclib_driver_t), intent(in) :: driver
      type(os_data_t), intent(in) :: os_data
      call os_system_call ("make compile " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
    end subroutine prclib_driver_make_compile
```

Combine all matrix-element code together with the driver in a process library
on disk.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: make_link => prclib_driver_make_link
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    subroutine prclib_driver_make_link (driver, os_data)
      class(prclib_driver_t), intent(in) :: driver
      type(os_data_t), intent(in) :: os_data
      call os_system_call ("make link " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
```

```
      end subroutine prclib_driver_make_link
```

14.1.15   Clean up generated files

The task of cleaning any generated files should also be deferred to Makefile
targets. Apart from removing everything, removing specific files may be useful
for partial rebuilds. (Note that removing the makefile itself can only be done
once, for obvious reasons.)

If there is no makefile, do nothing.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
      procedure :: clean_library => prclib_driver_clean_library
      procedure :: clean_objects => prclib_driver_clean_objects
      procedure :: clean_source => prclib_driver_clean_source
      procedure :: clean_driver => prclib_driver_clean_driver
      procedure :: clean_makefile => prclib_driver_clean_makefile
      procedure :: clean => prclib_driver_clean
      procedure :: distclean => prclib_driver_distclean
```

⟨*Prclib interfaces: procedures*⟩+≡
```
      subroutine prclib_driver_clean_library (driver, os_data)
        class(prclib_driver_t), intent(in) :: driver
        type(os_data_t), intent(in) :: os_data
        if (driver%makefile_exists ()) then
           call os_system_call ("make clean-library " // os_data%makeflags &
                // " -f " // driver%basename // ".makefile")
        end if
      end subroutine prclib_driver_clean_library

      subroutine prclib_driver_clean_objects (driver, os_data)
        class(prclib_driver_t), intent(in) :: driver
        type(os_data_t), intent(in) :: os_data
        if (driver%makefile_exists ()) then
           call os_system_call ("make clean-objects " // os_data%makeflags &
                // " -f " // driver%basename // ".makefile")
        end if
      end subroutine prclib_driver_clean_objects

      subroutine prclib_driver_clean_source (driver, os_data)
        class(prclib_driver_t), intent(in) :: driver
        type(os_data_t), intent(in) :: os_data
        if (driver%makefile_exists ()) then
           call os_system_call ("make clean-source " // os_data%makeflags &
                // " -f " // driver%basename // ".makefile")
        end if
      end subroutine prclib_driver_clean_source

      subroutine prclib_driver_clean_driver (driver, os_data)
        class(prclib_driver_t), intent(in) :: driver
        type(os_data_t), intent(in) :: os_data
        if (driver%makefile_exists ()) then
           call os_system_call ("make clean-driver " // os_data%makeflags &
                // " -f " // driver%basename // ".makefile")
```

```
      end if
   end subroutine prclib_driver_clean_driver

   subroutine prclib_driver_clean_makefile (driver, os_data)
     class(prclib_driver_t), intent(in) :: driver
     type(os_data_t), intent(in) :: os_data
     if (driver%makefile_exists ()) then
        call os_system_call ("make clean-makefile " // os_data%makeflags &
             // " -f " // driver%basename // ".makefile")
     end if
   end subroutine prclib_driver_clean_makefile

   subroutine prclib_driver_clean (driver, os_data)
     class(prclib_driver_t), intent(in) :: driver
     type(os_data_t), intent(in) :: os_data
     if (driver%makefile_exists ()) then
        call os_system_call ("make clean " // os_data%makeflags &
             // " -f " // driver%basename // ".makefile")
     end if
   end subroutine prclib_driver_clean

   subroutine prclib_driver_distclean (driver, os_data)
     class(prclib_driver_t), intent(in) :: driver
     type(os_data_t), intent(in) :: os_data
     if (driver%makefile_exists ()) then
        call os_system_call ("make distclean " // os_data%makeflags &
             // " -f " // driver%basename // ".makefile")
     end if
   end subroutine prclib_driver_distclean
```

This Make target should remove all files that apply to a specific process. We execute this when we want to force remaking source code. Note that source targets need not have prerequisites, so just calling `make_source` would not do anything if the files exist.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
   procedure :: clean_proc => prclib_driver_clean_proc
```

⟨*Prclib interfaces: procedures*⟩+≡
```
   subroutine prclib_driver_clean_proc (driver, i, os_data)
     class(prclib_driver_t), intent(in) :: driver
     integer, intent(in) :: i
     type(os_data_t), intent(in) :: os_data
     type(string_t) :: id
     if (driver%makefile_exists ()) then
        id = driver%record(i)%id
       call os_system_call ("make clean-" // driver%record(i)%id // " " &
             // os_data%makeflags &
             // " -f " // driver%basename // ".makefile")
     end if
   end subroutine prclib_driver_clean_proc
```

14.1.16 Further Tools

Check for the appropriate makefile.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: makefile_exists => prclib_driver_makefile_exists
```

⟨*Prclib interfaces: procedures*⟩+≡
```
function prclib_driver_makefile_exists (driver) result (flag)
  class(prclib_driver_t), intent(in) :: driver
  logical :: flag
  inquire (file = char (driver%basename) // ".makefile", exist = flag)
end function prclib_driver_makefile_exists
```

14.1.17 Load the library

Once the library has been linked, we can dlopen it and assign all procedure pointers to their proper places in the library driver object. The `loaded` flag is set only if all required pointers have become assigned.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: load => prclib_driver_load
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prclib_driver_load (driver, os_data, noerror)
  class(prclib_driver_t), intent(inout) :: driver
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: noerror
  type(c_funptr) :: c_fptr
  logical :: ignore

  ignore = .false.;  if (present (noerror))  ignore = noerror

  driver%libname = os_get_dlname (driver%basename, os_data, noerror, noerror)
  if (driver%libname == "")  return
  select type (driver)
  type is (prclib_driver_dynamic_t)
     if (.not. dlaccess_is_open (driver%dlaccess)) then
        call dlaccess_init &
             (driver%dlaccess, var_str ("."), driver%libname, os_data)
        if (.not. ignore)  call driver%check_dlerror ()
     end if
     driver%loaded = dlaccess_is_open (driver%dlaccess)
  class default
     driver%loaded = .true.
  end select
  if (.not. driver%loaded)  return

  c_fptr = driver%get_c_funptr (var_str ("get_n_processes"))
  call c_f_procpointer (c_fptr, driver%get_n_processes)
  driver%loaded = driver%loaded .and. associated (driver%get_n_processes)

  c_fptr = driver%get_c_funptr (var_str ("get_process_id_ptr"))
  call c_f_procpointer (c_fptr, driver%get_process_id_ptr)
  driver%loaded = driver%loaded .and. associated (driver%get_process_id_ptr)
```

```
c_fptr = driver%get_c_funptr (var_str ("get_model_name_ptr"))
call c_f_procpointer (c_fptr, driver%get_model_name_ptr)
driver%loaded = driver%loaded .and. associated (driver%get_model_name_ptr)

c_fptr = driver%get_c_funptr (var_str ("get_md5sum_ptr"))
call c_f_procpointer (c_fptr, driver%get_md5sum_ptr)
driver%loaded = driver%loaded .and. associated (driver%get_md5sum_ptr)

c_fptr = driver%get_c_funptr (var_str ("get_openmp_status"))
call c_f_procpointer (c_fptr, driver%get_openmp_status)
driver%loaded = driver%loaded .and. associated (driver%get_openmp_status)

c_fptr = driver%get_c_funptr (var_str ("get_n_in"))
call c_f_procpointer (c_fptr, driver%get_n_in)
driver%loaded = driver%loaded .and. associated (driver%get_n_in)

c_fptr = driver%get_c_funptr (var_str ("get_n_out"))
call c_f_procpointer (c_fptr, driver%get_n_out)
driver%loaded = driver%loaded .and. associated (driver%get_n_out)

c_fptr = driver%get_c_funptr (var_str ("get_n_flv"))
call c_f_procpointer (c_fptr, driver%get_n_flv)
driver%loaded = driver%loaded .and. associated (driver%get_n_flv)

c_fptr = driver%get_c_funptr (var_str ("get_n_hel"))
call c_f_procpointer (c_fptr, driver%get_n_hel)
driver%loaded = driver%loaded .and. associated (driver%get_n_hel)

c_fptr = driver%get_c_funptr (var_str ("get_n_col"))
call c_f_procpointer (c_fptr, driver%get_n_col)
driver%loaded = driver%loaded .and. associated (driver%get_n_col)

c_fptr = driver%get_c_funptr (var_str ("get_n_cin"))
call c_f_procpointer (c_fptr, driver%get_n_cin)
driver%loaded = driver%loaded .and. associated (driver%get_n_cin)

c_fptr = driver%get_c_funptr (var_str ("get_n_cf"))
call c_f_procpointer (c_fptr, driver%get_n_cf)
driver%loaded = driver%loaded .and. associated (driver%get_n_cf)

c_fptr = driver%get_c_funptr (var_str ("set_flv_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_flv_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_flv_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_hel_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_hel_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_hel_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_col_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_col_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_col_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_color_factors_ptr"))
```

```
          call c_f_procpointer (c_fptr, driver%set_color_factors_ptr)
          driver%loaded = driver%loaded .and. associated (driver%set_color_factors_ptr)

          c_fptr = driver%get_c_funptr (var_str ("get_fptr"))
          call c_f_procpointer (c_fptr, driver%get_fptr)
          driver%loaded = driver%loaded .and. associated (driver%get_fptr)

      end subroutine prclib_driver_load
```

Unload. To be sure, nullify the procedure pointers.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
      procedure :: unload => prclib_driver_unload
```

⟨*Prclib interfaces: procedures*⟩+≡
```
      subroutine prclib_driver_unload (driver)
        class(prclib_driver_t), intent(inout) :: driver
        select type (driver)
        type is (prclib_driver_dynamic_t)
           if (dlaccess_is_open (driver%dlaccess)) then
              call dlaccess_final (driver%dlaccess)
              call driver%check_dlerror ()
           end if
        end select
        driver%loaded = .false.
        nullify (driver%get_n_processes)
        nullify (driver%get_process_id_ptr)
        nullify (driver%get_model_name_ptr)
        nullify (driver%get_md5sum_ptr)
        nullify (driver%get_openmp_status)
        nullify (driver%get_n_in)
        nullify (driver%get_n_out)
        nullify (driver%get_n_flv)
        nullify (driver%get_n_hel)
        nullify (driver%get_n_col)
        nullify (driver%get_n_cin)
        nullify (driver%get_n_cf)
        nullify (driver%set_flv_state_ptr)
        nullify (driver%set_hel_state_ptr)
        nullify (driver%set_col_state_ptr)
        nullify (driver%set_color_factors_ptr)
        nullify (driver%get_fptr)
      end subroutine prclib_driver_unload
```

This subroutine checks the `dlerror` content and issues a fatal error if it finds
an error there.

⟨*Prclib interfaces: prclib driver dynamic: TBP*⟩≡
```
      procedure :: check_dlerror => prclib_driver_check_dlerror
```

⟨*Prclib interfaces: procedures*⟩+≡
```
      subroutine prclib_driver_check_dlerror (driver)
        class(prclib_driver_dynamic_t), intent(in) :: driver
        if (dlaccess_has_error (driver%dlaccess)) then
           call msg_fatal (char (dlaccess_get_error (driver%dlaccess)))
        end if
```

```
      end subroutine prclib_driver_check_dlerror
```

Get the handle (C function pointer) for a given "feature" of the matrix element code, so it can be assigned to the appropriate procedure pointer slot. In the static case, this is a trivial pointer assignment, hard-coded into the driver type implementation.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure (prclib_driver_get_c_funptr), deferred :: get_c_funptr
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
    abstract interface
      function prclib_driver_get_c_funptr (driver, feature) result (c_fptr)
        import
        class(prclib_driver_t), intent(inout) :: driver
        type(string_t), intent(in) :: feature
        type(c_funptr) :: c_fptr
      end function prclib_driver_get_c_funptr
    end interface
```

In the dynamic-library case, we call the DL interface to retrieve the C pointer to a named procedure.

⟨*Prclib interfaces: prclib driver dynamic: TBP*⟩+≡
```
    procedure :: get_c_funptr => prclib_driver_dynamic_get_c_funptr
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    function prclib_driver_dynamic_get_c_funptr (driver, feature) result (c_fptr)
      class(prclib_driver_dynamic_t), intent(inout) :: driver
      type(string_t), intent(in) :: feature
      type(c_funptr) :: c_fptr
      type(string_t) :: prefix, full_name
      prefix = driver%basename // "_"
      full_name = prefix // feature
      c_fptr = dlaccess_get_c_funptr (driver%dlaccess, full_name)
      call driver%check_dlerror ()
    end function prclib_driver_dynamic_get_c_funptr
```

14.1.18   MD5 sums

Recall the MD5 sum written in the Makefile

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: get_md5sum_makefile => prclib_driver_get_md5sum_makefile
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    function prclib_driver_get_md5sum_makefile (driver) result (md5sum)
      class(prclib_driver_t), intent(in) :: driver
      character(32) :: md5sum
      type(string_t) :: filename
      character(80) :: buffer
      logical :: exist
      integer :: u, iostat
      md5sum = ""
      filename = driver%basename // ".makefile"
```

```
      inquire (file = char (filename), exist = exist)
      if (exist) then
         u = free_unit ()
         open (u, file = char (filename), action = "read", status = "old")
         iostat = 0
         do
            read (u, "(A)", iostat = iostat)  buffer
            if (iostat /= 0)  exit
            buffer = adjustl (buffer)
            select case (buffer(1:9))
            case ("MD5SUM = ")
               read (buffer(11:), "(A32)")  md5sum
               exit
            end select
         end do
         close (u)
      end if
    end function prclib_driver_get_md5sum_makefile
```

Recall the MD5 sum written in the driver source code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
    procedure :: get_md5sum_driver => prclib_driver_get_md5sum_driver
```

⟨*Prclib interfaces: procedures*⟩+≡
```
    function prclib_driver_get_md5sum_driver (driver) result (md5sum)
      class(prclib_driver_t), intent(in) :: driver
      character(32) :: md5sum
      type(string_t) :: filename
      character(80) :: buffer
      logical :: exist
      integer :: u, iostat
      md5sum = ""
      filename = driver%basename // ".f90"
      inquire (file = char (filename), exist = exist)
      if (exist) then
         u = free_unit ()
         open (u, file = char (filename), action = "read", status = "old")
         iostat = 0
         do
            read (u, "(A)", iostat = iostat)  buffer
            if (iostat /= 0)  exit
            buffer = adjustl (buffer)
            select case (buffer(1:9))
            case ("md5sum = ")
               read (buffer(11:), "(A32)")  md5sum
               exit
            end select
         end do
         close (u)
      end if
    end function prclib_driver_get_md5sum_driver
```

Recall the MD5 sum written in the matrix element source code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
                procedure :: get_md5sum_source => prclib_driver_get_md5sum_source
⟨Prclib interfaces: procedures⟩+≡
        function prclib_driver_get_md5sum_source (driver, i) result (md5sum)
          class(prclib_driver_t), intent(in) :: driver
          integer, intent(in) :: i
          character(32) :: md5sum
          type(string_t) :: filename
          character(80) :: buffer
          logical :: exist
          integer :: u, iostat
          md5sum = ""

          filename = driver%record(i)%id // ".f90"
          inquire (file = char (filename), exist = exist)
          if (exist) then
             u = free_unit ()
             open (u, file = char (filename), action = "read", status = "old")
             iostat = 0
             do
                read (u, "(A)", iostat = iostat)  buffer
                if (iostat /= 0)  exit
                buffer = adjustl (buffer)
                select case (buffer(1:9))
                case ("md5sum = ")
                   read (buffer(11:), "(A32)")  md5sum
                   exit
                end select
             end do
             close (u)
          end if
        end function prclib_driver_get_md5sum_source
```

14.1.19   Test

This is the master for calling self-test procedures.

⟨*Prclib interfaces: public*⟩+≡
```
        public :: prclib_interfaces_test
```
⟨*Prclib interfaces: tests*⟩≡
```
        subroutine prclib_interfaces_test (u, results)
          integer, intent(in) :: u
          type(test_results_t), intent(inout) :: results
        ⟨Prclib interfaces: execute tests⟩
        end subroutine prclib_interfaces_test
```

Empty process list

Test 1: Create a driver object and display its contents. One of the feature lists
references a writer procedure; this is just a dummy that does nothing useful.

⟨*Prclib interfaces: execute tests*⟩≡
```
        call test (prclib_interfaces_1, "prclib_interfaces_1", &
```

```
                "create driver object", &
                u, results)
```

⟨*Prclib interfaces: tests*⟩+≡

```
    subroutine prclib_interfaces_1 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      character(32), parameter :: md5sum = "prclib_interfaces_1_md5sum      "
      class(prc_writer_t), pointer :: test_writer_1


      write (u, "(A)")  "* Test output: prclib_interfaces_1"
      write (u, "(A)")  "*   Purpose: display the driver object contents"
      write (u, *)
      write (u, "(A)")  "* Create a prclib driver object"
      write (u, "(A)")

      call dispatch_prclib_driver (driver, var_str ("prclib"), var_str (""))
      call driver%init (3)
      call driver%set_md5sum (md5sum)

      allocate (test_writer_1_t :: test_writer_1)

      call driver%set_record (1, var_str ("test1"), var_str ("test_model"), &
           [var_str ("init")], test_writer_1)

      call driver%set_record (2, var_str ("test2"), var_str ("foo_model"), &
           [var_str ("another_proc")], test_writer_1)

      call driver%set_record (3, var_str ("test3"), var_str ("test_model"), &
           [var_str ("init"), var_str ("some_proc")], test_writer_1)

      call driver%write (u)

      deallocate (test_writer_1)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_interfaces_1"
    end subroutine prclib_interfaces_1
```

The writer: the procedures write just comment lines. We can fix an instance
of this as a parameter (since it has no mutable content) and just reference the
fixed parameter.

NOTE: temporarily made public.

⟨*Prclib interfaces: test types*⟩≡

```
    type, extends (prc_writer_t) :: test_writer_1_t
     contains
       procedure, nopass :: type_name => test_writer_1_type_name
       procedure :: write_makefile_code => test_writer_1_mk
       procedure :: write_source_code => test_writer_1_src
       procedure :: write_interface => test_writer_1_if
       procedure :: write_md5sum_call => test_writer_1_md5sum
       procedure :: write_int_sub_call => test_writer_1_int_sub
       procedure :: write_col_state_call => test_writer_1_col_state
```

```
      procedure :: write_color_factors_call => test_writer_1_col_factors
    end type test_writer_1_t
```

⟨*Prclib interfaces: tests*⟩+≡

```
    function test_writer_1_type_name () result (string)
      type(string_t) :: string
      string = "test_1"
    end function test_writer_1_type_name

    subroutine test_writer_1_mk (writer, unit, id, os_data, testflag)
      class(test_writer_1_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: testflag
      write (unit, "(5A)")  "# Makefile code for process ", char (id), &
           " goes here."
    end subroutine test_writer_1_mk

    subroutine test_writer_1_src (writer, id)
      class(test_writer_1_t), intent(in) :: writer
      type(string_t), intent(in) :: id
    end subroutine test_writer_1_src

    subroutine test_writer_1_if (writer, unit, id, feature)
      class(test_writer_1_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, "(2x,9A)")  "! Interface code for ", &
          char (id), "_", char (writer%get_procname (feature)), &
          " goes here."
    end subroutine test_writer_1_if

    subroutine test_writer_1_md5sum (writer, unit, id)
      class(test_writer_1_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      write (unit, "(5x,9A)")  "! MD5sum call for ", char (id), " goes here."
    end subroutine test_writer_1_md5sum

    subroutine test_writer_1_int_sub (writer, unit, id, feature)
      class(test_writer_1_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, "(5x,9A)")  "! ", char (feature), " call for ", &
           char (id), " goes here."
    end subroutine test_writer_1_int_sub

    subroutine test_writer_1_col_state (writer, unit, id)
      class(test_writer_1_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      write (unit, "(5x,9A)")  "! col_state call for ", &
           char (id), " goes here."
```

1780

```
      end subroutine test_writer_1_col_state

    subroutine test_writer_1_col_factors (writer, unit, id)
      class(test_writer_1_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      write (unit, "(5x,9A)")  "! color_factors call for ", &
           char (id), " goes here."
    end subroutine test_writer_1_col_factors
```

Process library driver file

Test 2: Write the driver file for a test case with two processes. The first process needs no wrapper (C library), the second one needs wrappers (Fortran module library).

⟨*Prclib interfaces: execute tests*⟩+≡
```
    call test (prclib_interfaces_2, "prclib_interfaces_2", &
         "write driver file", &
         u, results)
```

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine prclib_interfaces_2 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      character(32), parameter :: md5sum = "prclib_interfaces_2_md5sum      "
      class(prc_writer_t), pointer :: test_writer_1, test_writer_2

      write (u, "(A)")  "* Test output: prclib_interfaces_2"
      write (u, "(A)")  "*   Purpose: check the generated driver source code"
      write (u, "(A)")
      write (u, "(A)")  "* Create a prclib driver object (2 processes)"
      write (u, "(A)")

      call dispatch_prclib_driver (driver, var_str ("prclib2"), var_str (""))
      call driver%init (2)
      call driver%set_md5sum (md5sum)

      allocate (test_writer_1_t :: test_writer_1)
      allocate (test_writer_2_t :: test_writer_2)

      call driver%set_record (1, var_str ("test1"), var_str ("Test_model"), &
           [var_str ("proc1")], test_writer_1)

      call driver%set_record (2, var_str ("test2"), var_str ("Test_model"), &
           [var_str ("proc1"), var_str ("proc2")], test_writer_2)

      call driver%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Write the driver file"
      write (u, "(A)")  "* File contents:"
      write (u, "(A)")
```

1781

```
      call driver%generate_driver_code (u)

      deallocate (test_writer_1)
      deallocate (test_writer_2)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_interfaces_2"
    end subroutine prclib_interfaces_2
```

A writer with wrapper code: the procedures again write just comment lines.
Since all procedures are NOPASS, we can reuse two of the TBP.

⟨*Prclib interfaces: test types*⟩+≡
```
    type, extends (prc_writer_f_module_t) :: test_writer_2_t
     contains
       procedure, nopass :: type_name => test_writer_2_type_name
       procedure :: write_makefile_code => test_writer_2_mk
       procedure :: write_source_code => test_writer_2_src
       procedure :: write_interface => test_writer_2_if
       procedure :: write_wrapper => test_writer_2_wr
    end type test_writer_2_t
```

⟨*Prclib interfaces: tests*⟩+≡
```
    function test_writer_2_type_name () result (string)
      type(string_t) :: string
      string = "test_2"
    end function test_writer_2_type_name

    subroutine test_writer_2_mk (writer, unit, id, os_data, testflag)
      class(test_writer_2_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: testflag
      write (unit, "(5A)")  "# Makefile code for process ", char (id), &
           " goes here."
    end subroutine test_writer_2_mk

    subroutine test_writer_2_src (writer, id)
      class(test_writer_2_t), intent(in) :: writer
      type(string_t), intent(in) :: id
    end subroutine test_writer_2_src

    subroutine test_writer_2_if (writer, unit, id, feature)
      class(test_writer_2_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, "(2x,9A)")  "! Interface code for ", &
         char (writer%get_module_name (id)), "_", &
         char (writer%get_procname (feature)), " goes here."
    end subroutine test_writer_2_if

    subroutine test_writer_2_wr (writer, unit, id, feature)
      class(test_writer_2_t), intent(in) :: writer
```

1782

```
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, *)
      write (unit, "(9A)")  "! Wrapper code for ", &
          char (writer%get_c_procname (id, feature)), " goes here."
    end subroutine test_writer_2_wr
```

Process library makefile

Test 3: Write the makefile for compiling and linking the process library (processes and driver code). There are two processes, one with one method, one with two methods.

To have predictable output, we reset the system-dependent initial components of `os_data` to known values.

⟨*Prclib interfaces: execute tests*⟩+≡
```
    call test (prclib_interfaces_3, "prclib_interfaces_3", &
        "write makefile", &
        u, results)
```

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine prclib_interfaces_3 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      type(os_data_t) :: os_data
      character(32), parameter :: md5sum = "prclib_interfaces_3_md5sum      "
      class(prc_writer_t), pointer :: test_writer_1, test_writer_2

      call os_data_init (os_data)
      os_data%fc = "fortran-compiler"
      os_data%whizard_includes = "-I module-dir"
      os_data%fcflags = "-C=all"
      os_data%fcflags_pic = "-PIC"
      os_data%cc = "c-compiler"
      os_data%cflags = "-I include-dir"
      os_data%cflags_pic = "-PIC"
      os_data%whizard_ldflags = ""
      os_data%ldflags = ""
      os_data%whizard_libtool = "my-libtool"
      os_data%latex = "latex -halt-on-error"
      os_data%mpost = "mpost --math=scaled -halt-on-error"
      os_data%dvips = "dvips"
      os_data%ps2pdf = "ps2pdf14"
      os_data%whizard_texpath = ""

      write (u, "(A)")  "* Test output: prclib_interfaces_3"
      write (u, "(A)")  "*   Purpose: check the generated Makefile"
      write (u, *)
      write (u, "(A)")  "* Create a prclib driver object (2 processes)"
      write (u, "(A)")

      call dispatch_prclib_driver (driver, var_str ("prclib3"), var_str (""))
      call driver%init (2)
      call driver%set_md5sum (md5sum)
```

```
      allocate (test_writer_1_t :: test_writer_1)
      allocate (test_writer_2_t :: test_writer_2)

      call driver%set_record (1, var_str ("test1"), var_str ("Test_model"), &
           [var_str ("proc1")], test_writer_1)

      call driver%set_record (2, var_str ("test2"), var_str ("Test_model"), &
           [var_str ("proc1"), var_str ("proc2")], test_writer_2)

      call driver%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Write Makefile"
      write (u, "(A)")  "* File contents:"
      write (u, "(A)")

      call driver%generate_makefile (u, os_data)

      deallocate (test_writer_1)
      deallocate (test_writer_2)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_interfaces_3"
    end subroutine prclib_interfaces_3
```

Compile test with Fortran module

Test 4: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a Fortran module, therefore we need a wrapper for the featured procedure.

⟨*Prclib interfaces: execute tests*⟩+≡
```
    call test (prclib_interfaces_4, "prclib_interfaces_4", &
         "compile and link (Fortran module)", &
         u, results)
```

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine prclib_interfaces_4 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      class(prc_writer_t), pointer :: test_writer_4
      type(os_data_t) :: os_data
      integer :: u_file

      integer, dimension(:,:), allocatable :: flv_state
      integer, dimension(:,:), allocatable :: hel_state
      integer, dimension(:,:,:), allocatable :: col_state
      logical, dimension(:,:), allocatable :: ghost_flag
      integer, dimension(:,:), allocatable :: cf_index
      complex(default), dimension(:), allocatable :: color_factors
      character(32), parameter :: md5sum = "prclib_interfaces_4_md5sum        "
      character(32) :: md5sum_file
```

1784

```
type(c_funptr) :: proc1_ptr
interface
   subroutine proc1_t (n) bind(C)
     import
     integer(c_int), intent(out) :: n
   end subroutine proc1_t
end interface
procedure(proc1_t), pointer :: proc1
integer(c_int) :: n

write (u, "(A)")  "* Test output: prclib_interfaces_4"
write (u, "(A)")  "*   Purpose: compile, link, and load process library"
write (u, "(A)")  "*           with (fake) matrix-element code &
     &as a Fortran module"
write (u, *)
write (u, "(A)")  "* Create a prclib driver object (1 process)"
write (u, "(A)")

call os_data_init (os_data)

allocate (test_writer_4_t :: test_writer_4)
call test_writer_4%init_test ()

call dispatch_prclib_driver (driver, var_str ("prclib4"), var_str (""))
call driver%init (1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test4"), var_str ("Test_model"), &
     [var_str ("proc1")], test_writer_4)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write Makefile"
u_file = free_unit ()
open (u_file, file="prclib4.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from Makefile"
write (u, "(A)")

md5sum_file = driver%get_md5sum_makefile ()
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Write driver source code"

u_file = free_unit ()
open (u_file, file="prclib4.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)
```

```
write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from driver source"
write (u, "(A)")

md5sum_file = driver%get_md5sum_driver ()
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from matrix-element source"
write (u, "(A)")

md5sum_file = driver%get_md5sum_source (1)
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
   write (u, "(A)")  "* Call library functions:"
   write (u, *)
   write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
   write (u, "(1x,A,A,A)")  "process_id    = '", &
        char (driver%get_process_id (1)), "'"
   write (u, "(1x,A,A,A)")  "model_name    = '", &
        char (driver%get_model_name (1)), "'"
   write (u, "(1x,A,A,A)")  "md5sum (lib)  = '", &
        char (driver%get_md5sum (0)), "'"
   write (u, "(1x,A,A,A)")  "md5sum (proc) = '", &
        char (driver%get_md5sum (1)), "'"
   write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
   write (u, "(1x,A,I0)")  "n_in  = ", driver%get_n_in (1)
   write (u, "(1x,A,I0)")  "n_out = ", driver%get_n_out (1)
   write (u, "(1x,A,I0)")  "n_flv = ", driver%get_n_flv (1)
   write (u, "(1x,A,I0)")  "n_hel = ", driver%get_n_hel (1)
   write (u, "(1x,A,I0)")  "n_col = ", driver%get_n_col (1)
   write (u, "(1x,A,I0)")  "n_cin = ", driver%get_n_cin (1)
   write (u, "(1x,A,I0)")  "n_cf  = ", driver%get_n_cf (1)
```

```
        call driver%set_flv_state (1, flv_state)
        write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

        call driver%set_hel_state (1, hel_state)
        write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

        call driver%set_col_state (1, col_state, ghost_flag)
        write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
        write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

        call driver%set_color_factors (1, color_factors, cf_index)
        write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
        write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

        call driver%get_fptr (1, 1, proc1_ptr)
        call c_f_procpointer (proc1_ptr, proc1)
        if (associated (proc1)) then
           write (u, *)
           call proc1 (n)
           write (u, "(1x,A,I0)")  "proc1(1) = ", n
        end if

     end if

     deallocate (test_writer_4)

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: prclib_interfaces_4"
   end subroutine prclib_interfaces_4
```

This version of test-code writer actually writes an interface and wrapper code. The wrapped function is a no-parameter function with integer result.

The stored MD5 sum may be modified.

We will reuse this later, therefore public.

⟨*Prclib interfaces: public*⟩+≡
```
   public :: test_writer_4_t
```

⟨*Prclib interfaces: test types*⟩+≡
```
   type, extends (prc_writer_f_module_t) :: test_writer_4_t
    contains
      procedure, nopass :: type_name => test_writer_4_type_name
      procedure, nopass :: get_module_name => &
           test_writer_4_get_module_name
      procedure :: write_makefile_code => test_writer_4_mk
      procedure :: write_source_code => test_writer_4_src
      procedure :: write_interface => test_writer_4_if
      procedure :: write_wrapper => test_writer_4_wr
   end type test_writer_4_t
```

⟨*Prclib interfaces: tests*⟩+≡
```
   function test_writer_4_type_name () result (string)
     type(string_t) :: string
     string = "test_4"
```

```
      end function test_writer_4_type_name

    function test_writer_4_get_module_name (id) result (name)
      type(string_t), intent(in) :: id
      type(string_t) :: name
      name = "tpr_" // id
    end function test_writer_4_get_module_name

    subroutine test_writer_4_mk (writer, unit, id, os_data, testflag)
      class(test_writer_4_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: testflag
      write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
      write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
      write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
      write (unit, "(5A)")  "CLEAN_OBJECTS += tpr_", char (id), ".mod"
      write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
      write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
      write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
    end subroutine test_writer_4_mk

    subroutine test_writer_4_src (writer, id)
      class(test_writer_4_t), intent(in) :: writer
      type(string_t), intent(in) :: id
      call write_test_module_file (id, var_str ("proc1"), writer%md5sum)
    end subroutine test_writer_4_src

    subroutine test_writer_4_if (writer, unit, id, feature)
      class(test_writer_4_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, "(2x,9A)")  "interface"
      write (unit, "(5x,9A)")  "subroutine ", &
         char (writer%get_c_procname (id, feature)), &
         " (n) bind(C)"
      write (unit, "(7x,9A)")  "import"
      write (unit, "(7x,9A)")  "implicit none"
      write (unit, "(7x,9A)")  "integer(c_int), intent(out) :: n"
      write (unit, "(5x,9A)")  "end subroutine ", &
         char (writer%get_c_procname (id, feature))
      write (unit, "(2x,9A)")  "end interface"
    end subroutine test_writer_4_if

    subroutine test_writer_4_wr (writer, unit, id, feature)
      class(test_writer_4_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      write (unit, *)
      write (unit, "(9A)")  "subroutine ", &
           char (writer%get_c_procname (id, feature)), &
         " (n) bind(C)"
      write (unit, "(2x,9A)")  "use iso_c_binding"
```

```
      write (unit, "(2x,9A)")  "use tpr_", char (id), ", only: ", &
          char (writer%get_procname (feature))
      write (unit, "(2x,9A)")  "implicit none"
      write (unit, "(2x,9A)")  "integer(c_int), intent(out) :: n"
      write (unit, "(2x,9A)")  "call ", char (feature), " (n)"
      write (unit, "(9A)")  "end subroutine ", &
          char (writer%get_c_procname (id, feature))
    end subroutine test_writer_4_wr
```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless $1 \to 2$ process, and it implements one additional function (feature), the name given as an argument.

⟨*Prclib interfaces: tests*⟩+≡

```
    subroutine write_test_module_file (basename, feature, md5sum)
      type(string_t), intent(in) :: basename
      type(string_t), intent(in) :: feature
      character(32), intent(in) :: md5sum
      integer :: u
      u = free_unit ()
      open (u, file = char (basename) // ".f90", &
          status = "replace", action = "write")
      write (u, "(A)")  "! (Pseudo) matrix element code file &
          &for WHIZARD self-test"
      write (u, *)
      write (u, "(A)")  "module tpr_" // char (basename)
      write (u, *)
      write (u, "(2x,A)")  "use kinds"
      write (u, "(2x,A)")  "use omega_color, OCF => omega_color_factor"
      write (u, *)
      write (u, "(2x,A)")  "implicit none"
      write (u, "(2x,A)")  "private"
      write (u, *)
      call write_test_me_code_1 (u)
      write (u, *)
      write (u, "(2x,A)")  "public :: " // char (feature)
      write (u, *)
      write (u, "(A)")  "contains"
      write (u, *)
      call write_test_me_code_2 (u, md5sum)
      write (u, *)
      write (u, "(2x,A)")  "subroutine " // char (feature) // " (n)"
      write (u, "(2x,A)")  "  integer, intent(out) :: n"
      write (u, "(2x,A)")  "  n = 42"
      write (u, "(2x,A)")  "end subroutine " // char (feature)
      write (u, *)
      write (u, "(A)")  "end module tpr_" // char (basename)
      close (u)
    end subroutine write_test_module_file
```

The following two subroutines provide building blocks for a matrix-element source code file, useful only for testing the workflow. The first routine writes the header part, the other routine the implementation of the procedures listed

in the header.

⟨*Prclib interfaces: tests*⟩+≡

```
subroutine write_test_me_code_1 (u)
  integer, intent(in) :: u
  write (u, "(2x,A)")  "public :: md5sum"
  write (u, "(2x,A)")  "public :: openmp_supported"
  write (u, *)
  write (u, "(2x,A)")  "public :: n_in"
  write (u, "(2x,A)")  "public :: n_out"
  write (u, "(2x,A)")  "public :: n_flv"
  write (u, "(2x,A)")  "public :: n_hel"
  write (u, "(2x,A)")  "public :: n_cin"
  write (u, "(2x,A)")  "public :: n_col"
  write (u, "(2x,A)")  "public :: n_cf"
  write (u, *)
  write (u, "(2x,A)")  "public :: flv_state"
  write (u, "(2x,A)")  "public :: hel_state"
  write (u, "(2x,A)")  "public :: col_state"
  write (u, "(2x,A)")  "public :: color_factors"
end subroutine write_test_me_code_1

subroutine write_test_me_code_2 (u, md5sum)
  integer, intent(in) :: u
  character(32), intent(in) :: md5sum
  write (u, "(2x,A)")  "pure function md5sum ()"
  write (u, "(2x,A)")  "  character(len=32) :: md5sum"
  write (u, "(2x,A)")  "  md5sum = '" // md5sum // "'"
  write (u, "(2x,A)")  "end function md5sum"
  write (u, *)
  write (u, "(2x,A)")  "pure function openmp_supported () result (status)"
  write (u, "(2x,A)")  "  logical :: status"
  write (u, "(2x,A)")  "  status = .false."
  write (u, "(2x,A)")  "end function openmp_supported"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_in () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_in"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_out () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 2"
  write (u, "(2x,A)")  "end function n_out"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_flv () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_flv"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_hel () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_hel"
  write (u, *)
```

```
      write (u, "(2x,A)")  "pure function n_cin () result (n)"
      write (u, "(2x,A)")  "  integer :: n"
      write (u, "(2x,A)")  "  n = 2"
      write (u, "(2x,A)")  "end function n_cin"
      write (u, *)
      write (u, "(2x,A)")  "pure function n_col () result (n)"
      write (u, "(2x,A)")  "  integer :: n"
      write (u, "(2x,A)")  "  n = 1"
      write (u, "(2x,A)")  "end function n_col"
      write (u, *)
      write (u, "(2x,A)")  "pure function n_cf () result (n)"
      write (u, "(2x,A)")  "  integer :: n"
      write (u, "(2x,A)")  "  n = 1"
      write (u, "(2x,A)")  "end function n_cf"
      write (u, *)
      write (u, "(2x,A)")  "pure subroutine flv_state (a)"
      write (u, "(2x,A)")  "  integer, dimension(:,:), intent(out) :: a"
      write (u, "(2x,A)")  "  a = reshape ([1,2,3], [3,1])"
      write (u, "(2x,A)")  "end subroutine flv_state"
      write (u, *)
      write (u, "(2x,A)")  "pure subroutine hel_state (a)"
      write (u, "(2x,A)")  "  integer, dimension(:,:), intent(out) :: a"
      write (u, "(2x,A)")  "  a = reshape ([0,0,0], [3,1])"
      write (u, "(2x,A)")  "end subroutine hel_state"
      write (u, *)
      write (u, "(2x,A)")  "pure subroutine col_state (a, g)"
      write (u, "(2x,A)")  "  integer, dimension(:,:,:), intent(out) :: a"
      write (u, "(2x,A)")  "  logical, dimension(:,:), intent(out) :: g"
      write (u, "(2x,A)")  "  a = reshape ([0,0, 0,0, 0,0], [2,3,1])"
      write (u, "(2x,A)")  "  g = reshape ([.false., .false., .false.], [3,1])"
      write (u, "(2x,A)")  "end subroutine col_state"
      write (u, *)
      write (u, "(2x,A)")  "pure subroutine color_factors (cf)"
      write (u, "(2x,A)")  "  type(OCF), dimension(:), intent(out) :: cf"
      write (u, "(2x,A)")  "  cf = [ OCF(1,1,+1._default) ]"
      write (u, "(2x,A)")  "end subroutine color_factors"
    end subroutine write_test_me_code_2
```

Compile test with Fortran bind(C) library

Test 5: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a Fortran library of independent procedures. These procedures are bind(C).

⟨*Prclib interfaces: execute tests*⟩+≡
```
    call test (prclib_interfaces_5, "prclib_interfaces_5", &
        "compile and link (Fortran library)", &
        u, results)
```

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine prclib_interfaces_5 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
```

1791

```
class(prc_writer_t), pointer :: test_writer_5
type(os_data_t) :: os_data
integer :: u_file

integer, dimension(:,:), allocatable :: flv_state
integer, dimension(:,:), allocatable :: hel_state
integer, dimension(:,:,:), allocatable :: col_state
logical, dimension(:,:), allocatable :: ghost_flag
integer, dimension(:,:), allocatable :: cf_index
complex(default), dimension(:), allocatable :: color_factors
character(32), parameter :: md5sum = "prclib_interfaces_5_md5sum      "

type(c_funptr) :: proc1_ptr
interface
   subroutine proc1_t (n) bind(C)
     import
     integer(c_int), intent(out) :: n
   end subroutine proc1_t
end interface
procedure(proc1_t), pointer :: proc1
integer(c_int) :: n

write (u, "(A)")  "* Test output: prclib_interfaces_5"
write (u, "(A)")  "*   Purpose: compile, link, and load process library"
write (u, "(A)")  "*            with (fake) matrix-element code &
     &as a Fortran bind(C) library"
write (u, *)
write (u, "(A)")  "* Create a prclib driver object (1 process)"
write (u, "(A)")

call os_data_init (os_data)
allocate (test_writer_5_t :: test_writer_5)

call dispatch_prclib_driver (driver, var_str ("prclib5"), var_str (""))
call driver%init (1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test5"), var_str ("Test_model"), &
     [var_str ("proc1")], test_writer_5)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib5.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib5.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)
```

```
write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
   write (u, "(A)")  "* Call library functions:"
   write (u, *)
   write (u, "(1x,A,I0)")  "n_processes  = ", driver%get_n_processes ()
   write (u, "(1x,A,A)")  "process_id   = ", &
        char (driver%get_process_id (1))
   write (u, "(1x,A,A)")  "model_name   = ", &
        char (driver%get_model_name (1))
   write (u, "(1x,A,A)")  "md5sum       = ", &
        char (driver%get_md5sum (1))
   write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
   write (u, "(1x,A,I0)")  "n_in  = ", driver%get_n_in (1)
   write (u, "(1x,A,I0)")  "n_out = ", driver%get_n_out (1)
   write (u, "(1x,A,I0)")  "n_flv = ", driver%get_n_flv (1)
   write (u, "(1x,A,I0)")  "n_hel = ", driver%get_n_hel (1)
   write (u, "(1x,A,I0)")  "n_col = ", driver%get_n_col (1)
   write (u, "(1x,A,I0)")  "n_cin = ", driver%get_n_cin (1)
   write (u, "(1x,A,I0)")  "n_cf  = ", driver%get_n_cf (1)

   call driver%set_flv_state (1, flv_state)
   write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

   call driver%set_hel_state (1, hel_state)
   write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

   call driver%set_col_state (1, col_state, ghost_flag)
   write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
   write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

   call driver%set_color_factors (1, color_factors, cf_index)
   write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
   write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

   call driver%get_fptr (1, 1, proc1_ptr)
   call c_f_procpointer (proc1_ptr, proc1)
   if (associated (proc1)) then
      write (u, *)
```

```
            call proc1 (n)
            write (u, "(1x,A,I0)")  "proc1(1) = ", n
         end if

      end if

      deallocate (test_writer_5)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_interfaces_5"
   end subroutine prclib_interfaces_5
```

This version of test-code writer writes interfaces for all standard features plus one specific feature. The interfaces are all bind(C), so no wrapper is needed.

⟨*Prclib interfaces: test types*⟩+≡

```
   type, extends (prc_writer_c_lib_t) :: test_writer_5_t
    contains
      procedure, nopass :: type_name => test_writer_5_type_name
      procedure :: write_makefile_code => test_writer_5_mk
      procedure :: write_source_code => test_writer_5_src
      procedure :: write_interface => test_writer_5_if
   end type test_writer_5_t
```

The

⟨*Prclib interfaces: tests*⟩+≡

```
   function test_writer_5_type_name () result (string)
     type(string_t) :: string
     string = "test_5"
   end function test_writer_5_type_name

   subroutine test_writer_5_mk (writer, unit, id, os_data, testflag)
     class(test_writer_5_t), intent(in) :: writer
     integer, intent(in) :: unit
     type(string_t), intent(in) :: id
     type(os_data_t), intent(in) :: os_data
     logical, intent(in), optional :: testflag
     write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
     write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
     write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
     write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
   end subroutine test_writer_5_mk

   subroutine test_writer_5_src (writer, id)
     class(test_writer_5_t), intent(in) :: writer
     type(string_t), intent(in) :: id
     call write_test_f_lib_file (id, var_str ("proc1"))
   end subroutine test_writer_5_src

   subroutine test_writer_5_if (writer, unit, id, feature)
     class(test_writer_5_t), intent(in) :: writer
     integer, intent(in) :: unit
     type(string_t), intent(in) :: id, feature
     select case (char (feature))
```

1794

```
      case ("proc1")
         write (unit, "(2x,9A)")  "interface"
         write (unit, "(5x,9A)")  "subroutine ", &
              char (writer%get_c_procname (id, feature)), &
              " (n) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "implicit none"
         write (unit, "(7x,9A)")  "integer(c_int), intent(out) :: n"
         write (unit, "(5x,9A)")  "end subroutine ", &
              char (writer%get_c_procname (id, feature))
         write (unit, "(2x,9A)")  "end interface"
      case default
         call writer%write_standard_interface (unit, id, feature)
      end select
    end subroutine test_writer_5_if
```

We need a test module file (actually, one for each process in the test above)
that allows us to check compilation and linking. The test module implements
a colorless $1 \rightarrow 2$ process, and it implements one additional function (feature),
the name given as an argument.

⟨*Prclib interfaces: tests*⟩+≡

```
    subroutine write_test_f_lib_file (basename, feature)
      type(string_t), intent(in) :: basename
      type(string_t), intent(in) :: feature
      integer :: u
      u = free_unit ()
      open (u, file = char (basename) // ".f90", &
            status = "replace", action = "write")
      write (u, "(A)")  "! (Pseudo) matrix element code file &
            &for WHIZARD self-test"
      call write_test_me_code_3 (u, char (basename))
      write (u, *)
      write (u, "(A)")  "subroutine " // char (basename) // "_" &
            // char (feature) // " (n) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  implicit none"
      write (u, "(A)")  "  integer(c_int), intent(out) :: n"
      write (u, "(A)")  "  n = 42"
      write (u, "(A)")  "end subroutine " // char (basename) // "_" &
            // char (feature)
      close (u)
    end subroutine write_test_f_lib_file
```

The following matrix-element source code is identical to the previous one, but
modified such as to provide independent procedures without a module envelope.

⟨*Prclib interfaces: tests*⟩+≡

```
    subroutine write_test_me_code_3 (u, id)
      integer, intent(in) :: u
      character(*), intent(in) :: id
      write (u, "(A)")  "function " // id // "_get_md5sum () &
            &result (cptr) bind(C)"
      write (u, "(A)")  "  use iso_c_binding"
      write (u, "(A)")  "  implicit none"
```

```
write (u, "(A)")  "  type(c_ptr) :: cptr"
write (u, "(A)")  "  character(c_char), dimension(32), &
     &target, save :: md5sum"
write (u, "(A)")  "  md5sum = copy (c_char_&
     &'1234567890abcdef1234567890abcdef')"
write (u, "(A)")  "  cptr = c_loc (md5sum)"
write (u, "(A)")  "contains"
write (u, "(A)")  "  function copy (md5sum)"
write (u, "(A)")  "    character(c_char), dimension(32) :: copy"
write (u, "(A)")  "    character(c_char), dimension(32), intent(in) :: &
     &md5sum"
write (u, "(A)")  "    copy = md5sum"
write (u, "(A)")  "  end function copy"
write (u, "(A)")  "end function " // id // "_get_md5sum"
write (u, *)
write (u, "(A)")  "function " // id // "_openmp_supported () &
     &result (status) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  logical(c_bool) :: status"
write (u, "(A)")  "  status = .false."
write (u, "(A)")  "end function " // id // "_openmp_supported"
write (u, *)
write (u, "(A)")  "function " // id // "_n_in () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_in"
write (u, *)
write (u, "(A)")  "function " // id // "_n_out () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 2"
write (u, "(A)")  "end function " // id // "_n_out"
write (u, *)
write (u, "(A)")  "function " // id // "_n_flv () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_flv"
write (u, *)
write (u, "(A)")  "function " // id // "_n_hel () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_hel"
write (u, *)
write (u, "(A)")  "function " // id // "_n_cin () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
```

```
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 2"
write (u, "(A)")  "end function " // id // "_n_cin"
write (u, *)
write (u, "(A)")  "function " // id // "_n_col () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_col"
write (u, *)
write (u, "(A)")  "function " // id // "_n_cf () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_cf"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_flv_state (flv_state) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: flv_state"
write (u, "(A)")  "  flv_state(1:3) = [1,2,3]"
write (u, "(A)")  "end subroutine " // id // "_flv_state"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_hel_state (hel_state) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: hel_state"
write (u, "(A)")  "  hel_state(1:3) = [0,0,0]"
write (u, "(A)")  "end subroutine " // id // "_hel_state"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_col_state &
    &(col_state, ghost_flag) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) &
    &:: col_state"
write (u, "(A)")  "  logical(c_bool), dimension(*), intent(out) &
    &:: ghost_flag"
write (u, "(A)")  "  col_state(1:6) = [0,0, 0,0, 0,0]"
write (u, "(A)")  "  ghost_flag(1:3) = [.false., .false., .false.]"
write (u, "(A)")  "end subroutine " // id // "_col_state"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_color_factors &
    &(cf_index1, cf_index2, color_factors) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  use kinds"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: cf_index1"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: cf_index2"
write (u, "(A)")  "  complex(c_default_complex), dimension(*), &
    &intent(out) :: color_factors"
write (u, "(A)")  "  cf_index1(1:1) = [1]"
```

```
      write (u, "(A)")  "  cf_index2(1:1) = [1]"
      write (u, "(A)")  "  color_factors(1:1) = [1]"
      write (u, "(A)")  "end subroutine " // id // "_color_factors"
    end subroutine write_test_me_code_3
```

Compile test with genuine C library

Test 6: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a C library of independent procedures. These procedures should match the Fortran bind(C) interface.

⟨*Prclib interfaces: execute tests*⟩+≡
```
    call test (prclib_interfaces_6, "prclib_interfaces_6", &
        "compile and link (C library)", &
        u, results)
```

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine prclib_interfaces_6 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      class(prc_writer_t), pointer :: test_writer_6
      type(os_data_t) :: os_data
      integer :: u_file

      integer, dimension(:,:), allocatable :: flv_state
      integer, dimension(:,:), allocatable :: hel_state
      integer, dimension(:,:,:), allocatable :: col_state
      logical, dimension(:,:), allocatable :: ghost_flag
      integer, dimension(:,:), allocatable :: cf_index
      complex(default), dimension(:), allocatable :: color_factors
      character(32), parameter :: md5sum = "prclib_interfaces_6_md5sum      "

      type(c_funptr) :: proc1_ptr
      interface
         subroutine proc1_t (n) bind(C)
           import
           integer(c_int), intent(out) :: n
         end subroutine proc1_t
      end interface
      procedure(proc1_t), pointer :: proc1
      integer(c_int) :: n

      write (u, "(A)")  "* Test output: prclib_interfaces_6"
      write (u, "(A)")  "*   Purpose: compile, link, and load process library"
      write (u, "(A)")  "*            with (fake) matrix-element code &
          &as a C library"
      write (u, *)
      write (u, "(A)")  "* Create a prclib driver object (1 process)"
      write (u, "(A)")

      call os_data_init (os_data)
      allocate (test_writer_6_t :: test_writer_6)
```

1798

```
call dispatch_prclib_driver (driver, var_str ("prclib6"), var_str (""))
call driver%init (1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test6"), var_str ("Test_model"), &
     [var_str ("proc1")], test_writer_6)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib6.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib6.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
   write (u, "(A)")  "* Call library functions:"
   write (u, *)
   write (u, "(1x,A,I0)")  "n_processes  = ", driver%get_n_processes ()
   write (u, "(1x,A,A)")   "process_id   = ", &
        char (driver%get_process_id (1))
   write (u, "(1x,A,A)")   "model_name   = ", &
        char (driver%get_model_name (1))
   write (u, "(1x,A,A)")   "md5sum       = ", &
        char (driver%get_md5sum (1))
   write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
   write (u, "(1x,A,I0)")  "n_in  = ", driver%get_n_in (1)
   write (u, "(1x,A,I0)")  "n_out = ", driver%get_n_out (1)
   write (u, "(1x,A,I0)")  "n_flv = ", driver%get_n_flv (1)
   write (u, "(1x,A,I0)")  "n_hel = ", driver%get_n_hel (1)
   write (u, "(1x,A,I0)")  "n_col = ", driver%get_n_col (1)
```

1799

```
        write (u, "(1x,A,I0)")  "n_cin = ", driver%get_n_cin (1)
        write (u, "(1x,A,I0)")  "n_cf  = ", driver%get_n_cf (1)

        call driver%set_flv_state (1, flv_state)
        write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

        call driver%set_hel_state (1, hel_state)
        write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

        call driver%set_col_state (1, col_state, ghost_flag)
        write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
        write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

        call driver%set_color_factors (1, color_factors, cf_index)
        write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
        write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

        call driver%get_fptr (1, 1, proc1_ptr)
        call c_f_procpointer (proc1_ptr, proc1)
        if (associated (proc1)) then
           write (u, *)
           call proc1 (n)
           write (u, "(1x,A,I0)")  "proc1(1) = ", n
        end if

     end if

     deallocate (test_writer_6)

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: prclib_interfaces_6"
  end subroutine prclib_interfaces_6
```

This version of test-code writer writes interfaces for all standard features plus one specific feature. The interfaces are all bind(C), so no wrapper is needed.

The driver part is identical to the Fortran case, so we simply extend the previous `test_writer_5` type. We only have to override the Makefile writer.

⟨*Prclib interfaces: test types*⟩+≡
```
     type, extends (test_writer_5_t) :: test_writer_6_t
      contains
        procedure, nopass :: type_name => test_writer_6_type_name
        procedure :: write_makefile_code => test_writer_6_mk
        procedure :: write_source_code => test_writer_6_src
     end type test_writer_6_t
```

⟨*Prclib interfaces: tests*⟩+≡
```
     function test_writer_6_type_name () result (string)
       type(string_t) :: string
       string = "test_6"
     end function test_writer_6_type_name

     subroutine test_writer_6_mk (writer, unit, id, os_data, testflag)
       class(test_writer_6_t), intent(in) :: writer
```

```
        integer, intent(in) :: unit
        type(string_t), intent(in) :: id
        type(os_data_t), intent(in) :: os_data
        logical, intent(in), optional :: testflag
        write (unit, "(5A)")  "SOURCES += ", char (id), ".c"
        write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
        write (unit, "(5A)")  char (id), ".lo: ", char (id), ".c"
        write (unit, "(5A)")  TAB, "$(LTCCOMPILE) $<"
      end subroutine test_writer_6_mk

      subroutine test_writer_6_src (writer, id)
        class(test_writer_6_t), intent(in) :: writer
        type(string_t), intent(in) :: id
        call write_test_c_lib_file (id, var_str ("proc1"))
      end subroutine test_writer_6_src
```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless $1 \to 2$ process, and it implements one additional function (feature), the name given as an argument.

⟨*Prclib interfaces: tests*⟩+≡
```
      subroutine write_test_c_lib_file (basename, feature)
        type(string_t), intent(in) :: basename
        type(string_t), intent(in) :: feature
        integer :: u
        u = free_unit ()
        open (u, file = char (basename) // ".c", &
             status = "replace", action = "write")
        write (u, "(A)")  "/* (Pseudo) matrix element code file &
             &for WHIZARD self-test */"
        write (u, "(A)")  "#include <stdbool.h>"
        write (u, *)
        call write_test_me_code_4 (u, char (basename))
        write (u, *)
        write (u, "(A)")  "void " // char (basename) // "_" &
             // char (feature) // "(int* n) {"
        write (u, "(A)")  "  *n = 42;"
        write (u, "(A)")  "}"
        close (u)
      end subroutine write_test_c_lib_file
```

The following matrix-element source code is equivalent to the code in the previous example, but coded in C.

⟨*Prclib interfaces: tests*⟩+≡
```
      subroutine write_test_me_code_4 (u, id)
        integer, intent(in) :: u
        character(*), intent(in) :: id
        write (u, "(A)")  "char* " // id // "_get_md5sum() {"
        write (u, "(A)")  "  return ""1234567890abcdef1234567890abcdef"";"
        write (u, "(A)")  "}"
        write (u, *)
        write (u, "(A)")  "bool " // id // "_openmp_supported() {"
        write (u, "(A)")  "  return false;"
```

```
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_in() {"
write (u, "(A)")  "  return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_out() {"
write (u, "(A)")  "  return 2;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_flv() {"
write (u, "(A)")  "  return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_hel() {"
write (u, "(A)")  "  return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_cin() {"
write (u, "(A)")  "  return 2;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_col() {"
write (u, "(A)")  "  return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "int " // id // "_n_cf() {"
write (u, "(A)")  "  return 1;"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "void " // id // "_flv_state( int (*a)[] ) {"
write (u, "(A)")  "  static int flv_state[1][3] =  { { 1, 2, 3 } };"
write (u, "(A)")  "  int j;"
write (u, "(A)")  "  for (j = 0; j < 3; j++) { (*a)[j] &
     &= flv_state[0][j]; }"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "void " // id // "_hel_state( int (*a)[] ) {"
write (u, "(A)")  "  static int hel_state[1][3] =  { { 0, 0, 0 } };"
write (u, "(A)")  "  int j;"
write (u, "(A)")  "  for (j = 0; j < 3; j++) { (*a)[j] &
     &= hel_state[0][j]; }"
write (u, "(A)")  "}"
write (u, *)
write (u, "(A)")  "void " // id // "_col_state&
     &( int (*a)[], bool (*g)[] ) {"
write (u, "(A)")  "  static int col_state[1][3][2] = &
     &{ { {0, 0}, {0, 0}, {0, 0} } };"
write (u, "(A)")  "  static bool ghost_flag[1][3] =  &
     &{ { false, false, false } };"
write (u, "(A)")  "  int j,k;"
write (u, "(A)")  "  for (j = 0; j < 3; j++) {"
write (u, "(A)")  "    for (k = 0; k < 2; k++) {"
write (u, "(A)")  "       (*a)[j*2+k] = col_state[0][j][k];"
```

```
      write (u, "(A)")  "     }"
      write (u, "(A)")  "     (*g)[j] = ghost_flag[0][j];"
      write (u, "(A)")  "  }"
      write (u, "(A)")  "}"
      write (u, *)
      if (c_default_complex == c_long_double_complex) then
         write (u, "(A)")  "void " // id // "_color_factors&
            &( int (*cf_index1)[], int (*cf_index2)[], &
            &long double _Complex (*color_factors)[] ) {"
      else
         write (u, "(A)")  "void " // id // "_color_factors&
            &( int (*cf_index1)[], int (*cf_index2)[], &
            &double _Complex (*color_factors)[] ) {"
      end if
      write (u, "(A)")  "  (*color_factors)[0] = 1;"
      write (u, "(A)")  "  (*cf_index1)[0] = 1;"
      write (u, "(A)")  "  (*cf_index2)[0] = 1;"
      write (u, "(A)")  "}"
    end subroutine write_test_me_code_4
```

Test cleanup targets

Test 7: Repeat test 4 (create, compile, link Fortran module and driver) and
properly clean up all generated files.

⟨*Prclib interfaces: execute tests*⟩+≡
```
    call test (prclib_interfaces_7, "prclib_interfaces_7", &
         "cleanup", &
         u, results)
```

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine prclib_interfaces_7 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      class(prc_writer_t), pointer :: test_writer_4
      type(os_data_t) :: os_data
      integer :: u_file
      character(32), parameter :: md5sum = "1234567890abcdef1234567890abcdef"

      write (u, "(A)")  "* Test output: prclib_interfaces_7"
      write (u, "(A)")  "*   Purpose: compile and link process library"
      write (u, "(A)")  "*            with (fake) matrix-element code &
           &as a Fortran module"
      write (u, "(A)")  "*            then clean up generated files"
      write (u, *)
      write (u, "(A)")  "* Create a prclib driver object (1 process)"

      allocate (test_writer_4_t :: test_writer_4)

      call os_data_init (os_data)
      call dispatch_prclib_driver (driver, var_str ("prclib7"), var_str (""))
      call driver%init (1)
      call driver%set_md5sum (md5sum)
      call driver%set_record (1, var_str ("test7"), var_str ("Test_model"), &
```

```
      [var_str ("proc1")], test_writer_4)

write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib7.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib7.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)


write (u, "(A)")  "* File check"
write (u, *)
call check_file (u, "test7.f90")
call check_file (u, "tpr_test7.mod")
call check_file (u, "test7.lo")
call check_file (u, "prclib7.makefile")
call check_file (u, "prclib7.f90")
call check_file (u, "prclib7.lo")
call check_file (u, "prclib7.la")

write (u, *)
write (u, "(A)")  "* Delete library"
write (u, *)
call driver%clean_library (os_data)
call check_file (u, "prclib7.la")

write (u, *)
write (u, "(A)")  "* Delete object code"
write (u, *)
call driver%clean_objects (os_data)
call check_file (u, "test7.lo")
call check_file (u, "tpr_test7.mod")
call check_file (u, "prclib7.lo")

write (u, *)
write (u, "(A)")  "* Delete source code"
write (u, *)
call driver%clean_source (os_data)
call check_file (u, "test7.f90")
```

```
      write (u, *)
      write (u, "(A)")  "* Delete driver source code"
      write (u, *)
      call driver%clean_driver (os_data)
      call check_file (u, "prclib7.f90")

      write (u, *)
      write (u, "(A)")  "* Delete makefile"
      write (u, *)
      call driver%clean_makefile (os_data)
      call check_file (u, "prclib7.makefile")

      deallocate (test_writer_4)

      write (u, *)
      write (u, "(A)")  "* Test output end: prclib_interfaces_7"
    end subroutine prclib_interfaces_7
```

Auxiliary routine: check and report existence of a file

⟨*Prclib interfaces: tests*⟩+≡
```
    subroutine check_file (u, file)
      integer, intent(in) :: u
      character(*), intent(in) :: file
      logical :: exist
      inquire (file=file, exist=exist)
      write (u, "(2x,A,A,L1)")  file, " = ", exist
    end subroutine check_file
```

14.2 Abstract process core configuration

In this module, we define abstract data types that handle the method-specific part of defining a process (including all of its options) and accessing an external matrix element.

There are no unit tests, these are deferred to the `process_libraries` module below.

⟨`prc_core_def.f90`⟩≡
  ⟨*File header*⟩

  module prc_core_def

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!

    use process_constants
    use prclib_interfaces

  ⟨*Standard module head*⟩

  ⟨*Prc core def: public*⟩

⟨*Prc core def: types*⟩

⟨*Prc core def: interfaces*⟩

```
contains
```

⟨*Prc core def: procedures*⟩

```
end module prc_core_def
```

14.2.1 Process core definition type

For storing configuration data that depend on the specific process variant, we introduce a polymorphic type. At this point, we just declare an abstract base type. This allows us to defer the implementation to later modules.

There should be no components that need explicit finalization, otherwise we would have to call a finalizer from the `process_component_def_t` wrapper.

⟨*Prc core def: public*⟩≡
```
    public :: prc_core_def_t
```

⟨*Prc core def: types*⟩≡
```
    type, abstract :: prc_core_def_t
      class(prc_writer_t), allocatable :: writer
     contains
     ⟨Prc core def: process core def: TBP⟩
    end type prc_core_def_t
```

Interfaces for the deferred methods.

This returns a string. No passed argument; the string is constant and depends just on the type.

⟨*Prc core def: process core def: TBP*⟩≡
```
    procedure (prc_core_def_get_string), nopass, deferred :: type_string
```

⟨*Prc core def: interfaces*⟩≡
```
    abstract interface
      function prc_core_def_get_string () result (string)
        import
        type(string_t) :: string
      end function prc_core_def_get_string
    end interface
```

The `write` method should display the content completely.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure (prc_core_def_write), deferred :: write
```

⟨*Prc core def: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_core_def_write (object, unit)
        import
        class(prc_core_def_t), intent(in) :: object
        integer, intent(in) :: unit
      end subroutine prc_core_def_write
```

```
      end interface
```

The `read` method should fill the content completely.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure (prc_core_def_read), deferred :: read
```

⟨*Prc core def: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_core_def_read (object, unit)
         import
         class(prc_core_def_t), intent(out) :: object
         integer, intent(in) :: unit
       end subroutine prc_core_def_read
    end interface
```

This communicates a MD5 checksum to the writer inside the `core_def` object, if there is any. Usually, this checksum is not yet known at the time when the writer is initialized.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure :: set_md5sum => prc_core_def_set_md5sum
```

⟨*Prc core def: procedures*⟩≡
```
    subroutine prc_core_def_set_md5sum (core_def, md5sum)
      class(prc_core_def_t), intent(inout) :: core_def
      character(32) :: md5sum
      if (allocated (core_def%writer))  core_def%writer%md5sum = md5sum
    end subroutine prc_core_def_set_md5sum
```

Allocate an appropriate driver object which corresponds to the chosen process core definition.

For internal matrix element (i.e., those which do not need external code), the driver should have access to all matrix element information from the beginning. In short, it is the matrix-element code.

For external matrix elements, the driver will get access to the external matrix element code.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure(prc_core_def_allocate_driver), deferred :: allocate_driver
```

⟨*Prc core def: interfaces*⟩+≡
```
    interface
       subroutine prc_core_def_allocate_driver (object, driver, basename)
         import
         class(prc_core_def_t), intent(in) :: object
         class(prc_core_driver_t), intent(out), allocatable :: driver
         type(string_t), intent(in) :: basename
       end subroutine prc_core_def_allocate_driver
    end interface
```

This flag tells whether the particular variant needs external code. We implement a default function which returns false. The flag depends only on the type, therefore we implement it as `nopass`.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure, nopass :: needs_code => prc_core_def_needs_code
```

1807

⟨*Prc core def: procedures*⟩+≡
```
    function prc_core_def_needs_code () result (flag)
      logical :: flag
      flag = .false.
    end function prc_core_def_needs_code
```

This subroutine allocates an array which holds the name of all features that this process core implements. This feature applies to matrix element code that is not coded as a Fortran module but communicates via independent library functions, which follow the C calling conventions. The addresses of those functions are returned as C function pointers, which can be converted into Fortran procedure pointers. The conversion is done in code specific for the process variant; here we just retrieve the C function pointer.

The array returned here serves the purpose of writing specific driver code. The driver interfaces only those C functions which are supported for the given process core.

If the process core does not require external code, this array is meaningless.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure(prc_core_def_get_features), nopass, deferred &
        :: get_features
```

⟨*Prc core def: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_core_def_get_features (features)
        import
        type(string_t), dimension(:), allocatable, intent(out) :: features
      end subroutine prc_core_def_get_features
    end interface
```

Assign pointers to the process-specific procedures to the driver, if the process is external.

⟨*Prc core def: process core def: TBP*⟩+≡
```
    procedure(prc_core_def_connect), deferred :: connect
```

⟨*Prc core def: interfaces*⟩+≡
```
    abstract interface
      subroutine prc_core_def_connect (def, lib_driver, i, proc_driver)
        import
        class(prc_core_def_t), intent(in) :: def
        class(prclib_driver_t), intent(in) :: lib_driver
        integer, intent(in) :: i
        class(prc_core_driver_t), intent(inout) :: proc_driver
      end subroutine prc_core_def_connect
    end interface
```

14.2.2 Process core template

We must be able to automatically allocate a process core definition object with the appropriate type, given only the type name.

To this end, we introduce a `prc_template_t` type which is simply a wrapper for an empty `prc_core_def_t` object. Choosing one of the templates from an array, we can allocate the target object.

⟨*Prc core def: public*⟩+≡
```
public :: prc_template_t
```

⟨*Prc core def: types*⟩+≡
```
type :: prc_template_t
   class(prc_core_def_t), allocatable :: core_def
end type prc_template_t
```

The allocation routine. We use the `source` option of the `allocate` statement. The `mold` option would probably more appropriate, but is a F2008 feature.

⟨*Prc core def: public*⟩+≡
```
public :: allocate_core_def
```

⟨*Prc core def: procedures*⟩+≡
```
subroutine allocate_core_def (template, name, core_def)
  type(prc_template_t), dimension(:), intent(in) :: template
  type(string_t), intent(in) :: name
  class(prc_core_def_t), allocatable :: core_def
  integer :: i
  do i = 1, size (template)
     if (template(i)%core_def%type_string () == name) then
        allocate (core_def, source = template(i)%core_def)
        return
     end if
  end do
end subroutine allocate_core_def
```

14.2.3 Process driver

For each process component, we implement a driver object which holds the calls to the matrix element and various auxiliary routines as procedure pointers. Any actual calculation will use this object to communicate with the process.

Depending on the type of process (as described by a corresponding `prc_core_def` object), the procedure pointers may refer to external or internal code, and there may be additional procedures for certain types. The base type defined here is abstract.

⟨*Prc core def: public*⟩+≡
```
public :: prc_core_driver_t
```

⟨*Prc core def: types*⟩+≡
```
type, abstract :: prc_core_driver_t
 contains
 ⟨Prc core def: process driver: TBP⟩
end type prc_core_driver_t
```

This returns the process type. No reference to contents.

⟨*Prc core def: process driver: TBP*⟩≡
```
procedure(prc_core_driver_type_name), nopass, deferred :: type_name
```

⟨*Prc core def: interfaces*⟩+≡
```
    abstract interface
       function prc_core_driver_type_name () result (type)
         import
         type(string_t) :: type
       end function prc_core_driver_type_name
    end interface
```

14.2.4   Process driver for intrinsic process

This is an abstract extension for the driver type. It has one additional method, namely a subroutine that fills the record of constant process data. For an external process, this task is performed by the external library driver instead.

⟨*Prc core def: public*⟩+≡
```
    public :: process_driver_internal_t
```

⟨*Prc core def: types*⟩+≡
```
    type, extends (prc_core_driver_t), abstract :: process_driver_internal_t
     contains
     ⟨Prc core def: process driver internal: TBP⟩
    end type process_driver_internal_t
```

⟨*Prc core def: process driver internal: TBP*⟩≡
```
    procedure(process_driver_fill_constants), deferred :: fill_constants
```

⟨*Prc core def: interfaces*⟩+≡
```
    abstract interface
       subroutine process_driver_fill_constants (driver, data)
         import
         class(process_driver_internal_t), intent(in) :: driver
         type(process_constants_t), intent(out) :: data
       end subroutine process_driver_fill_constants
    end interface
```

14.3   Particle Specifiers

In this module we introduce a type for specifying a particle or particle alternative. In addition to the particle specifiers (strings separated by colons), the type contains an optional flag `polarized` and a string `decay`. If the `polarized` flag is set, particle polarization information should be kept when generating events for this process. If the `decay` string is set, it is the ID of a decay process which should be applied to this particle when generating events.

In input/output form, the `polarized` flag is indicated by an asterisk (`*`) in brackets, and the `decay` is indicated by its ID in brackets.

The `read` and `write` procedures in this module are not type-bound but generic procedures which handle scalar and array arguments.

⟨`particle_specifiers.f90`⟩≡
  ⟨*File header*⟩

```
module particle_specifiers

⟨Use strings⟩
⟨Use file utils⟩
  use diagnostics !NODEP!
  use unit_tests

⟨Standard module head⟩

⟨Particle specifiers: public⟩

⟨Particle specifiers: types⟩

⟨Particle specifiers: interfaces⟩

contains

⟨Particle specifiers: procedures⟩

⟨Particle specifiers: tests⟩

end module particle_specifiers
```

14.3.1 Base type

This is an abstract type which can hold a single particle or an expression.

⟨*Particle specifiers: types*⟩≡
```
    type, abstract :: prt_spec_expr_t
     contains
     ⟨Particle specifiers: prt spec expr: TBP⟩
    end type prt_spec_expr_t
```

Output, as a string.

⟨*Particle specifiers: prt spec expr: TBP*⟩≡
```
    procedure (prt_spec_expr_to_string), deferred :: to_string
```

⟨*Particle specifiers: interfaces*⟩≡
```
    abstract interface
       function prt_spec_expr_to_string (object) result (string)
         import
         class(prt_spec_expr_t), intent(in) :: object
         type(string_t) :: string
       end function prt_spec_expr_to_string
    end interface
```

Call an `expand` method for all enclosed subexpressions (before handling the current expression).

⟨*Particle specifiers: prt spec expr: TBP*⟩+≡
```
    procedure (prt_spec_expr_expand_sub), deferred :: expand_sub
```

⟨*Particle specifiers: interfaces*⟩+≡
```
    abstract interface
       subroutine prt_spec_expr_expand_sub (object)
```

```
        import
        class(prt_spec_expr_t), intent(inout) :: object
      end subroutine prt_spec_expr_expand_sub
    end interface
```

14.3.2   Wrapper type

This wrapper can hold a particle expression of any kind. We need it so we can
make variadic arrays.

⟨*Particle specifiers: public*⟩≡
```
    public :: prt_expr_t
```

⟨*Particle specifiers: types*⟩+≡
```
    type :: prt_expr_t
      class(prt_spec_expr_t), allocatable :: x
     contains
```
⟨*Particle specifiers: prt expr: TBP*⟩
```
    end type prt_expr_t
```

Output as a string: delegate.

⟨*Particle specifiers: prt expr: TBP*⟩≡
```
    procedure :: to_string => prt_expr_to_string
```

⟨*Particle specifiers: procedures*⟩≡
```
    recursive function prt_expr_to_string (object) result (string)
      class(prt_expr_t), intent(in) :: object
      type(string_t) :: string
      if (allocated (object%x)) then
         string = object%x%to_string ()
      else
         string = ""
      end if
    end function prt_expr_to_string
```

Allocate the expression as a particle specifier and copy the value.

⟨*Particle specifiers: prt expr: TBP*⟩+≡
```
    procedure :: init_spec => prt_expr_init_spec
```

⟨*Particle specifiers: procedures*⟩+≡
```
    subroutine prt_expr_init_spec (object, spec)
      class(prt_expr_t), intent(out) :: object
      type(prt_spec_t), intent(in) :: spec
      allocate (prt_spec_t :: object%x)
      select type (x => object%x)
      type is (prt_spec_t)
         x = spec
      end select
    end subroutine prt_expr_init_spec
```

Allocate as a list/sum and allocate for a given length

⟨*Particle specifiers: prt expr: TBP*⟩+≡
```
    procedure :: init_list => prt_expr_init_list
    procedure :: init_sum => prt_expr_init_sum
```

⟨*Particle specifiers: procedures*⟩+≡
```
subroutine prt_expr_init_list (object, n)
  class(prt_expr_t), intent(out) :: object
  integer, intent(in) :: n
  allocate (prt_spec_list_t :: object%x)
  select type (x => object%x)
  type is (prt_spec_list_t)
     allocate (x%expr (n))
  end select
end subroutine prt_expr_init_list

subroutine prt_expr_init_sum (object, n)
  class(prt_expr_t), intent(out) :: object
  integer, intent(in) :: n
  allocate (prt_spec_sum_t :: object%x)
  select type (x => object%x)
  type is (prt_spec_sum_t)
     allocate (x%expr (n))
  end select
end subroutine prt_expr_init_sum
```

Return the number of terms. This is unity, except if the expression is a sum.

⟨*Particle specifiers: prt expr: TBP*⟩+≡
```
procedure :: get_n_terms => prt_expr_get_n_terms
```

⟨*Particle specifiers: procedures*⟩+≡
```
function prt_expr_get_n_terms (object) result (n)
  class(prt_expr_t), intent(in) :: object
  integer :: n
  if (allocated (object%x)) then
     select type (x => object%x)
     type is (prt_spec_sum_t)
        n = size (x%expr)
     class default
        n = 1
     end select
  else
     n = 0
  end if
end function prt_expr_get_n_terms
```

Transform one of the terms, as returned by the previous method, to an array of particle specifiers. The array has more than one entry if the selected term is a list. This makes sense only if the expression has been completely expanded, so the list contains only atoms.

⟨*Particle specifiers: prt expr: TBP*⟩+≡
```
procedure :: term_to_array => prt_expr_term_to_array
```

⟨*Particle specifiers: procedures*⟩+≡
```
recursive subroutine prt_expr_term_to_array (object, array, i)
  class(prt_expr_t), intent(in) :: object
  type(prt_spec_t), dimension(:), intent(inout), allocatable :: array
  integer, intent(in) :: i
  integer :: j
```

```
      if (allocated (array))  deallocate (array)
      select type (x => object%x)
      type is (prt_spec_t)
         allocate (array (1))
         array(1) = x
      type is (prt_spec_list_t)
         allocate (array (size (x%expr)))
         do j = 1, size (array)
            select type (y => x%expr(j)%x)
            type is (prt_spec_t)
               array(j) = y
            end select
         end do
      type is (prt_spec_sum_t)
         call x%expr(i)%term_to_array (array, 1)
      end select
    end subroutine prt_expr_term_to_array
```

14.3.3   The atomic type

The trivial case is a single particle, including optional decay and polarization attributes.

Definition

The particle is unstable if the `decay` array is allocated. The `polarized` flag and decays may not be set simultaneously.

⟨*Particle specifiers: public*⟩+≡
```
    public :: prt_spec_t
```

⟨*Particle specifiers: types*⟩+≡
```
    type, extends (prt_spec_expr_t) :: prt_spec_t
      private
      type(string_t) :: name
      logical :: polarized = .false.
      type(string_t), dimension(:), allocatable :: decay
     contains
     ⟨Particle specifiers: prt spec: TBP⟩
    end type prt_spec_t
```

I/O

Output. Old-style subroutines.

⟨*Particle specifiers: public*⟩+≡
```
    public :: prt_spec_write
```

⟨*Particle specifiers: interfaces*⟩+≡
```
    interface prt_spec_write
       module procedure prt_spec_write1
       module procedure prt_spec_write2
    end interface prt_spec_write
```

⟨*Particle specifiers: procedures*⟩+≡
```
    subroutine prt_spec_write1 (object, unit, advance)
      type(prt_spec_t), intent(in) :: object
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: advance
      character(3) :: adv
      integer :: u
      u = output_unit (unit)
      adv = "yes";  if (present (advance))  adv = advance
      write (u, "(A)", advance = adv)  char (object%to_string ())
    end subroutine prt_spec_write1
```

Write an array as a list of particle specifiers.

⟨*Particle specifiers: procedures*⟩+≡
```
    subroutine prt_spec_write2 (prt_spec, unit, advance)
      type(prt_spec_t), dimension(:), intent(in) :: prt_spec
      integer, intent(in), optional :: unit
      character(len=*), intent(in), optional :: advance
      character(3) :: adv
      integer :: u, i
      u = output_unit (unit)
      adv = "yes"; if (present (advance))  adv = advance
      do i = 1, size (prt_spec)
         if (i > 1)  write (u, "(A)", advance="no")  ", "
         call prt_spec_write (prt_spec(i), u, advance="no")
      end do
      write (u, "(A)", advance = adv)
    end subroutine prt_spec_write2
```

Read. Input may be string or array of strings.

⟨*Particle specifiers: public*⟩+≡
```
    public :: prt_spec_read
```

⟨*Particle specifiers: interfaces*⟩+≡
```
    interface prt_spec_read
       module procedure prt_spec_read1
       module procedure prt_spec_read2
    end interface prt_spec_read
```

Read a single particle specifier

⟨*Particle specifiers: procedures*⟩+≡
```
    pure subroutine prt_spec_read1 (prt_spec, string)
      type(prt_spec_t), intent(out) :: prt_spec
      type(string_t), intent(in) :: string
      type(string_t) :: arg, buffer
      integer :: b1, b2, c, n, i
      b1 = scan (string, "(")
      b2 = scan (string, ")")
      if (b1 == 0) then
         prt_spec%name = trim (adjustl (string))
      else
         prt_spec%name = trim (adjustl (extract (string, 1, b1-1)))
         arg = trim (adjustl (extract (string, b1+1, b2-1)))
         if (arg == "*") then
```

```
        prt_spec%polarized = .true.
      else
         n = 0
         buffer = arg
         do
            if (verify (buffer, " ") == 0)  exit
            n = n + 1
            c = scan (buffer, "+")
            if (c == 0)  exit
            buffer = extract (buffer, c+1)
         end do
         allocate (prt_spec%decay (n))
         buffer = arg
         do i = 1, n
            c = scan (buffer, "+")
            if (c == 0)  c = len (buffer) + 1
            prt_spec%decay(i) = trim (adjustl (extract (buffer, 1, c-1)))
            buffer = extract (buffer, c+1)
         end do
      end if
    end if
  end subroutine prt_spec_read1
```

Read a particle specifier array, given as a single string. The array is allocated
to the correct size.

⟨*Particle specifiers: procedures*⟩+≡

```
  pure subroutine prt_spec_read2 (prt_spec, string)
    type(prt_spec_t), dimension(:), intent(out), allocatable :: prt_spec
    type(string_t), intent(in) :: string
    type(string_t) :: buffer
    integer :: c, i, n
    n = 0
    buffer = string
    do
       n = n + 1
       c = scan (buffer, ",")
       if (c == 0)  exit
       buffer = extract (buffer, c+1)
    end do
    allocate (prt_spec (n))
    buffer = string
    do i = 1, size (prt_spec)
       c = scan (buffer, ",")
       if (c == 0)  c = len (buffer) + 1
       call prt_spec_read (prt_spec(i), &
            trim (adjustl (extract (buffer, 1, c-1))))
       buffer = extract (buffer, c+1)
    end do
  end subroutine prt_spec_read2
```

Constructor

Initialize a particle specifier.

⟨*Particle specifiers: public*⟩+≡
```
public :: new_prt_spec
```

⟨*Particle specifiers: interfaces*⟩+≡
```
interface new_prt_spec
   module procedure new_prt_spec
   module procedure new_prt_spec_polarized
   module procedure new_prt_spec_unstable
end interface new_prt_spec
```

⟨*Particle specifiers: procedures*⟩+≡
```
elemental function new_prt_spec (name) result (prt_spec)
  type(string_t), intent(in) :: name
  type(prt_spec_t) :: prt_spec
  prt_spec%name = name
end function new_prt_spec

elemental function new_prt_spec_polarized (name, polarized) result (prt_spec)
  type(string_t), intent(in) :: name
  logical, intent(in) :: polarized
  type(prt_spec_t) :: prt_spec
  prt_spec%name = name
  prt_spec%polarized = polarized
end function new_prt_spec_polarized

pure function new_prt_spec_unstable (name, decay) result (prt_spec)
  type(string_t), intent(in) :: name
  type(string_t), dimension(:), intent(in) :: decay
  type(prt_spec_t) :: prt_spec
  prt_spec%name = name
  allocate (prt_spec%decay (size (decay)))
  prt_spec%decay = decay
end function new_prt_spec_unstable
```

Access Methods

Return the particle name without qualifiers

⟨*Particle specifiers: prt spec: TBP*⟩≡
```
procedure :: get_name => prt_spec_get_name
```

⟨*Particle specifiers: procedures*⟩+≡
```
elemental function prt_spec_get_name (prt_spec) result (name)
  class(prt_spec_t), intent(in) :: prt_spec
  type(string_t) :: name
  name = prt_spec%name
end function prt_spec_get_name
```

Return the name with qualifiers

⟨*Particle specifiers: prt spec: TBP*⟩+≡
```
procedure :: to_string => prt_spec_to_string
```

⟨*Particle specifiers: procedures*⟩+≡
```
    function prt_spec_to_string (object) result (string)
      class(prt_spec_t), intent(in) :: object
      type(string_t) :: string
      integer :: i
      string = object%name
      if (allocated (object%decay)) then
         string = string // "("
         do i = 1, size (object%decay)
            if (i > 1)  string = string // " + "
            string = string // object%decay(i)
         end do
         string = string // ")"
      else if (object%polarized) then
         string = string // "(*)"
      end if
    end function prt_spec_to_string
```

Return the polarization flag

⟨*Particle specifiers: prt spec: TBP*⟩+≡
```
    procedure :: is_polarized => prt_spec_is_polarized
```
⟨*Particle specifiers: procedures*⟩+≡
```
    elemental function prt_spec_is_polarized (prt_spec) result (flag)
      class(prt_spec_t), intent(in) :: prt_spec
      logical :: flag
      flag = prt_spec%polarized
    end function prt_spec_is_polarized
```

The particle is unstable if there is a decay array.

⟨*Particle specifiers: prt spec: TBP*⟩+≡
```
    procedure :: is_unstable => prt_spec_is_unstable
```
⟨*Particle specifiers: procedures*⟩+≡
```
    elemental function prt_spec_is_unstable (prt_spec) result (flag)
      class(prt_spec_t), intent(in) :: prt_spec
      logical :: flag
      flag = allocated (prt_spec%decay)
    end function prt_spec_is_unstable
```

Return the number of decay channels

⟨*Particle specifiers: prt spec: TBP*⟩+≡
```
    procedure :: get_n_decays => prt_spec_get_n_decays
```
⟨*Particle specifiers: procedures*⟩+≡
```
    elemental function prt_spec_get_n_decays (prt_spec) result (n)
      class(prt_spec_t), intent(in) :: prt_spec
      integer :: n
      if (allocated (prt_spec%decay)) then
         n = size (prt_spec%decay)
      else
         n = 0
      end if
    end function prt_spec_get_n_decays
```

Return the decay channels

⟨Particle specifiers: prt spec: TBP⟩+≡
```
    procedure :: get_decays => prt_spec_get_decays
```

⟨Particle specifiers: procedures⟩+≡
```
    subroutine prt_spec_get_decays (prt_spec, decay)
      class(prt_spec_t), intent(in) :: prt_spec
      type(string_t), dimension(:), allocatable, intent(out) :: decay
      if (allocated (prt_spec%decay)) then
          allocate (decay (size (prt_spec%decay)))
          decay = prt_spec%decay
      else
          allocate (decay (0))
      end if
    end subroutine prt_spec_get_decays
```

Miscellaneous

There is nothing to expand here:

⟨Particle specifiers: prt spec: TBP⟩+≡
```
    procedure :: expand_sub => prt_spec_expand_sub
```

⟨Particle specifiers: procedures⟩+≡
```
    subroutine prt_spec_expand_sub (object)
      class(prt_spec_t), intent(inout) :: object
    end subroutine prt_spec_expand_sub
```

14.3.4   List

A list of particle specifiers, indicating, e.g., the final state of a process.

⟨Particle specifiers: public⟩+≡
```
    public :: prt_spec_list_t
```

⟨Particle specifiers: types⟩+≡
```
    type, extends (prt_spec_expr_t) :: prt_spec_list_t
      type(prt_expr_t), dimension(:), allocatable :: expr
     contains
```
⟨Particle specifiers: prt spec list: TBP⟩
```
    end type prt_spec_list_t
```

Output: Concatenate the components. Insert brackets if the component is also a list. The components of the `expr` array, if any, should all be filled.

⟨Particle specifiers: prt spec list: TBP⟩≡
```
    procedure :: to_string => prt_spec_list_to_string
```

⟨Particle specifiers: procedures⟩+≡
```
    recursive function prt_spec_list_to_string (object) result (string)
      class(prt_spec_list_t), intent(in) :: object
      type(string_t) :: string
      integer :: i
      string = ""
      if (allocated (object%expr)) then
```

```
        do i = 1, size (object%expr)
           if (i > 1)  string = string // ", "
           select type (x => object%expr(i)%x)
           type is (prt_spec_list_t)
              string = string // "(" // x%to_string () // ")"
           class default
              string = string // x%to_string ()
           end select
        end do
     end if
  end function prt_spec_list_to_string
```

Flatten: if there is a subexpression which is also a list, include the components as direct members of the current list.

⟨*Particle specifiers: prt spec list: TBP*⟩+≡
```
     procedure :: flatten => prt_spec_list_flatten
```

⟨*Particle specifiers: procedures*⟩+≡
```
  subroutine prt_spec_list_flatten (object)
    class(prt_spec_list_t), intent(inout) :: object
    type(prt_expr_t), dimension(:), allocatable :: tmp_expr
    integer :: i, n_flat, i_flat
    n_flat = 0
    do i = 1, size (object%expr)
       select type (y => object%expr(i)%x)
       type is (prt_spec_list_t)
          n_flat = n_flat + size (y%expr)
       class default
          n_flat = n_flat + 1
       end select
    end do
    if (n_flat > size (object%expr)) then
       allocate (tmp_expr (n_flat))
       i_flat = 0
       do i = 1, size (object%expr)
          select type (y => object%expr(i)%x)
          type is (prt_spec_list_t)
             tmp_expr (i_flat + 1 : i_flat + size (y%expr)) = y%expr
             i_flat = i_flat + size (y%expr)
          class default
             tmp_expr (i_flat + 1) = object%expr(i)
             i_flat = i_flat + 1
          end select
       end do
    end if
    if (allocated (tmp_expr)) &
         call move_alloc (from = tmp_expr, to = object%expr)
  end subroutine prt_spec_list_flatten
```

Convert a list of sums into a sum of lists. (Subexpressions which are not sums are left untouched.)

⟨*Particle specifiers: procedures*⟩+≡
```
  subroutine distribute_prt_spec_list (object)
```

```fortran
      class(prt_spec_expr_t), intent(inout), allocatable :: object
      class(prt_spec_expr_t), allocatable :: new_object
      integer, dimension(:), allocatable :: n, ii
      integer :: k, n_expr, n_terms, i_term
      select type (object)
      type is (prt_spec_list_t)
         n_expr = size (object%expr)
         allocate (n (n_expr), source = 1)
         allocate (ii (n_expr), source = 1)
         do k = 1, size (object%expr)
            select type (y => object%expr(k)%x)
            type is (prt_spec_sum_t)
               n(k) = size (y%expr)
            end select
         end do
         n_terms = product (n)
         if (n_terms > 1) then
            allocate (prt_spec_sum_t :: new_object)
            select type (new_object)
            type is (prt_spec_sum_t)
               allocate (new_object%expr (n_terms))
               do i_term = 1, n_terms
                  allocate (prt_spec_list_t :: new_object%expr(i_term)%x)
                  select type (x => new_object%expr(i_term)%x)
                  type is (prt_spec_list_t)
                     allocate (x%expr (n_expr))
                     do k = 1, n_expr
                        select type (y => object%expr(k)%x)
                        type is (prt_spec_sum_t)
                           x%expr(k) = y%expr(ii(k))
                        class default
                           x%expr(k) = object%expr(k)
                        end select
                     end do
                  end select
                  INCR_INDEX: do k = n_expr, 1, -1
                     if (ii(k) < n(k)) then
                        ii(k) = ii(k) + 1
                        exit INCR_INDEX
                     else
                        ii(k) = 1
                     end if
                  end do INCR_INDEX
               end do
            end select
         end if
      end select
      if (allocated (new_object)) call move_alloc (from = new_object, to = object)
    end subroutine distribute_prt_spec_list
```

Apply **expand** to all components of the list.

⟨*Particle specifiers: prt spec list: TBP*⟩+≡
```fortran
    procedure :: expand_sub => prt_spec_list_expand_sub
```

⟨*Particle specifiers: procedures*⟩+≡
```
recursive subroutine prt_spec_list_expand_sub (object)
  class(prt_spec_list_t), intent(inout) :: object
  integer :: i
  if (allocated (object%expr)) then
     do i = 1, size (object%expr)
        call object%expr(i)%expand ()
     end do
  end if
end subroutine prt_spec_list_expand_sub
```

14.3.5 Sum

A sum of particle specifiers, indicating, e.g., a sum of final states.

⟨*Particle specifiers: public*⟩+≡
```
public :: prt_spec_sum_t
```

⟨*Particle specifiers: types*⟩+≡
```
type, extends (prt_spec_expr_t) :: prt_spec_sum_t
  type(prt_expr_t), dimension(:), allocatable :: expr
 contains
 ⟨Particle specifiers: prt spec sum: TBP⟩
end type prt_spec_sum_t
```

Output: Concatenate the components. Insert brackets if the component is a list or also a sum. The components of the `expr` array, if any, should all be filled.

⟨*Particle specifiers: prt spec sum: TBP*⟩≡
```
procedure :: to_string => prt_spec_sum_to_string
```

⟨*Particle specifiers: procedures*⟩+≡
```
recursive function prt_spec_sum_to_string (object) result (string)
  class(prt_spec_sum_t), intent(in) :: object
  type(string_t) :: string
  integer :: i
  string = ""
  if (allocated (object%expr)) then
     do i = 1, size (object%expr)
        if (i > 1)  string = string // " + "
        select type (x => object%expr(i)%x)
        type is (prt_spec_list_t)
           string = string // "(" // x%to_string () // ")"
        type is (prt_spec_sum_t)
           string = string // "(" // x%to_string () // ")"
        class default
           string = string // x%to_string ()
        end select
     end do
  end if
end function prt_spec_sum_to_string
```

Flatten: if there is a subexpression which is also a sum, include the components as direct members of the current sum.

This is identical to `prt_spec_list_flatten` above, except for the type.

⟨*Particle specifiers: prt spec sum: TBP*⟩+≡

```
    procedure :: flatten => prt_spec_sum_flatten
```

⟨*Particle specifiers: procedures*⟩+≡

```
    subroutine prt_spec_sum_flatten (object)
      class(prt_spec_sum_t), intent(inout) :: object
      type(prt_expr_t), dimension(:), allocatable :: tmp_expr
      integer :: i, n_flat, i_flat
      n_flat = 0
      do i = 1, size (object%expr)
         select type (y => object%expr(i)%x)
         type is (prt_spec_sum_t)
            n_flat = n_flat + size (y%expr)
            class default
            n_flat = n_flat + 1
         end select
      end do
      if (n_flat > size (object%expr)) then
         allocate (tmp_expr (n_flat))
         i_flat = 0
         do i = 1, size (object%expr)
            select type (y => object%expr(i)%x)
            type is (prt_spec_sum_t)
               tmp_expr (i_flat + 1 : i_flat + size (y%expr)) = y%expr
               i_flat = i_flat + size (y%expr)
            class default
               tmp_expr (i_flat + 1) = object%expr(i)
               i_flat = i_flat + 1
            end select
         end do
      end if
      if (allocated (tmp_expr)) &
           call move_alloc (from = tmp_expr, to = object%expr)
    end subroutine prt_spec_sum_flatten
```

Apply `expand` to all terms in the sum.

⟨*Particle specifiers: prt spec sum: TBP*⟩+≡

```
    procedure :: expand_sub => prt_spec_sum_expand_sub
```

⟨*Particle specifiers: procedures*⟩+≡

```
    recursive subroutine prt_spec_sum_expand_sub (object)
      class(prt_spec_sum_t), intent(inout) :: object
      integer :: i
      if (allocated (object%expr)) then
         do i = 1, size (object%expr)
            call object%expr(i)%expand ()
         end do
      end if
    end subroutine prt_spec_sum_expand_sub
```

14.3.6 Expression Expansion

The `expand` method transforms each particle specifier expression into a sum of lists, according to the rules

$$a, (b, c) \to a, b, c \tag{14.1}$$
$$a + (b + c) \to a + b + c \tag{14.2}$$
$$a, b + c \to (a, b) + (a, c) \tag{14.3}$$

Note that the precedence of comma and plus are opposite to this expansion, so the parentheses in the final expression are necessary.

We assume that subexpressions are filled, i.e., arrays are allocated.

⟨*Particle specifiers: prt expr: TBP*⟩+≡
```
    procedure :: expand => prt_expr_expand
```

⟨*Particle specifiers: procedures*⟩+≡
```
    recursive subroutine prt_expr_expand (expr)
      class(prt_expr_t), intent(inout) :: expr
      if (allocated (expr%x)) then
         call distribute_prt_spec_list (expr%x)
         call expr%x%expand_sub ()
         select type (x => expr%x)
         type is (prt_spec_list_t)
            call x%flatten ()
         type is (prt_spec_sum_t)
            call x%flatten ()
         end select
      end if
    end subroutine prt_expr_expand
```

14.3.7 Test

This is the master for calling self-test procedures.

⟨*Particle specifiers: public*⟩+≡
```
    public :: particle_specifiers_test
```

⟨*Particle specifiers: tests*⟩≡
```
    subroutine particle_specifiers_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Particle specifiers: execute tests*⟩
```
    end subroutine particle_specifiers_test
```

Particle specifier array

Define, read and write an array of particle specifiers.

⟨*Particle specifiers: execute tests*⟩≡
```
    call test (particle_specifiers_1, "particle_specifiers_1", &
        "Handle particle specifiers", &
        u, results)
```

⟨*Particle specifiers: tests*⟩+≡

```
    subroutine particle_specifiers_1 (u)
      integer, intent(in) :: u
      type(prt_spec_t), dimension(:), allocatable :: prt_spec
      type(string_t), dimension(:), allocatable :: decay
      type(string_t), dimension(0) :: no_decay
      integer :: i, j

      write (u, "(A)")  "* Test output: particle_specifiers_1"
      write (u, "(A)")  "*   Purpose: Read and write a particle specifier array"
      write (u, "(A)")

      allocate (prt_spec (5))
      prt_spec = [ &
          new_prt_spec (var_str ("a")), &
          new_prt_spec (var_str ("b"), .true.), &
          new_prt_spec (var_str ("c"), [var_str ("dec1")]), &
          new_prt_spec (var_str ("d"), [var_str ("dec1"), var_str ("dec2")]), &
          new_prt_spec (var_str ("e"), no_decay) &
          ]
      do i = 1, size (prt_spec)
         write (u, "(A)")  char (prt_spec(i)%to_string ())
      end do
      write (u, "(A)")

      call prt_spec_read (prt_spec, &
          var_str (" a, b( *), c( dec1), d (dec1 + dec2 ), e()"))
      call prt_spec_write (prt_spec, u)

      do i = 1, size (prt_spec)
         write (u, "(A)")
         write (u, "(A,A)")  char (prt_spec(i)%get_name ()), ":"
         write (u, "(A,L1)")  "polarized = ", prt_spec(i)%is_polarized ()
         write (u, "(A,L1)")  "unstable  = ", prt_spec(i)%is_unstable ()
         write (u, "(A,I0)")  "n_decays  = ", prt_spec(i)%get_n_decays ()
         call prt_spec(i)%get_decays (decay)
         write (u, "(A)", advance="no") "decays    ="
         do j = 1, size (decay)
            write (u, "(1x,A)", advance="no") char (decay(j))
         end do
         write (u, "(A)")
      end do

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: particle_specifiers_1"
    end subroutine particle_specifiers_1
```

Particle specifier expressions

Nested expressions (only basic particles, no decay specs).

⟨*Particle specifiers: execute tests*⟩+≡

```
    call test (particle_specifiers_2, "particle_specifiers_2", &
```

```
                "Particle specifier expressions", &
                u, results)
⟨Particle specifiers: tests⟩+≡
     subroutine particle_specifiers_2 (u)
       integer, intent(in) :: u
       type(prt_spec_t) :: a, b, c, d, e, f
       type(prt_expr_t) :: pe1, pe2, pe3
       type(prt_expr_t) :: pe4, pe5, pe6, pe7, pe8, pe9
       integer :: i
       type(prt_spec_t), dimension(:), allocatable :: pa

       write (u, "(A)")  "* Test output: particle_specifiers_2"
       write (u, "(A)")  "*   Purpose: Create and display particle expressions"
       write (u, "(A)")

       write (u, "(A)")  "* Basic expressions"
       write (u, *)

       a = new_prt_spec (var_str ("a"))
       b = new_prt_spec (var_str ("b"))
       c = new_prt_spec (var_str ("c"))
       d = new_prt_spec (var_str ("d"))
       e = new_prt_spec (var_str ("e"))
       f = new_prt_spec (var_str ("f"))

       call pe1%init_spec (a)
       write (u, "(A)")  char (pe1%to_string ())

       call pe2%init_sum (2)
       select type (x => pe2%x)
       type is (prt_spec_sum_t)
          call x%expr(1)%init_spec (a)
          call x%expr(2)%init_spec (b)
       end select
       write (u, "(A)")  char (pe2%to_string ())

       call pe3%init_list (2)
       select type (x => pe3%x)
       type is (prt_spec_list_t)
          call x%expr(1)%init_spec (a)
          call x%expr(2)%init_spec (b)
       end select
       write (u, "(A)")  char (pe3%to_string ())

       write (u, *)
       write (u, "(A)")  "* Nested expressions"
       write (u, *)

       call pe4%init_list (2)
       select type (x => pe4%x)
       type is (prt_spec_list_t)
          call x%expr(1)%init_sum (2)
          select type (y => x%expr(1)%x)
          type is (prt_spec_sum_t)
```

```
         call y%expr(1)%init_spec (a)
         call y%expr(2)%init_spec (b)
      end select
      call x%expr(2)%init_spec (c)
end select
write (u, "(A)")  char (pe4%to_string ())

call pe5%init_list (2)
select type (x => pe5%x)
type is (prt_spec_list_t)
   call x%expr(1)%init_list (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_list_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_spec (b)
   end select
   call x%expr(2)%init_spec (c)
end select
write (u, "(A)")  char (pe5%to_string ())

call pe6%init_sum (2)
select type (x => pe6%x)
type is (prt_spec_sum_t)
   call x%expr(1)%init_spec (a)
   call x%expr(2)%init_sum (2)
   select type (y => x%expr(2)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (b)
      call y%expr(2)%init_spec (c)
   end select
end select
write (u, "(A)")  char (pe6%to_string ())

call pe7%init_list (2)
select type (x => pe7%x)
type is (prt_spec_list_t)
   call x%expr(1)%init_sum (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_list (2)
      select type (z => y%expr(2)%x)
      type is (prt_spec_list_t)
         call z%expr(1)%init_spec (b)
         call z%expr(2)%init_spec (c)
      end select
   end select
   call x%expr(2)%init_spec (d)
end select
write (u, "(A)")  char (pe7%to_string ())

call pe8%init_sum (2)
select type (x => pe8%x)
type is (prt_spec_sum_t)
```

```
      call x%expr(1)%init_list (2)
      select type (y => x%expr(1)%x)
      type is (prt_spec_list_t)
         call y%expr(1)%init_spec (a)
         call y%expr(2)%init_spec (b)
      end select
      call x%expr(2)%init_list (2)
      select type (y => x%expr(2)%x)
      type is (prt_spec_list_t)
         call y%expr(1)%init_spec (c)
         call y%expr(2)%init_spec (d)
      end select
end select
write (u, "(A)")  char (pe8%to_string ())

call pe9%init_list (3)
select type (x => pe9%x)
type is (prt_spec_list_t)
      call x%expr(1)%init_sum (2)
      select type (y => x%expr(1)%x)
      type is (prt_spec_sum_t)
         call y%expr(1)%init_spec (a)
         call y%expr(2)%init_spec (b)
      end select
      call x%expr(2)%init_spec (c)
      call x%expr(3)%init_sum (3)
      select type (y => x%expr(3)%x)
      type is (prt_spec_sum_t)
         call y%expr(1)%init_spec (d)
         call y%expr(2)%init_spec (e)
         call y%expr(3)%init_spec (f)
      end select
end select
write (u, "(A)")  char (pe9%to_string ())

write (u, *)
write (u, "(A)")  "* Expand as sum"
write (u, *)

call pe1%expand ()
write (u, "(A)")  char (pe1%to_string ())

call pe4%expand ()
write (u, "(A)")  char (pe4%to_string ())

call pe5%expand ()
write (u, "(A)")  char (pe5%to_string ())

call pe6%expand ()
write (u, "(A)")  char (pe6%to_string ())

call pe7%expand ()
write (u, "(A)")  char (pe7%to_string ())
```

```
      call pe8%expand ()
      write (u, "(A)")  char (pe8%to_string ())

      call pe9%expand ()
      write (u, "(A)")  char (pe9%to_string ())

      write (u, *)
      write (u, "(A)")  "* Transform to arrays:"

      write (u, "(A)")  "* Atomic specifier"
      do i = 1, pe1%get_n_terms ()
         call pe1%term_to_array (pa, i)
         call prt_spec_write (pa, u)
      end do

      write (u, *)
      write (u, "(A)")  "* List"
      do i = 1, pe5%get_n_terms ()
         call pe5%term_to_array (pa, i)
         call prt_spec_write (pa, u)
      end do

      write (u, *)
      write (u, "(A)")  "* Sum of atoms"
      do i = 1, pe6%get_n_terms ()
         call pe6%term_to_array (pa, i)
         call prt_spec_write (pa, u)
      end do

      write (u, *)
      write (u, "(A)")  "* Sum of lists"
      do i = 1, pe9%get_n_terms ()
         call pe9%term_to_array (pa, i)
         call prt_spec_write (pa, u)
      end do

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: particle_specifiers_2"
    end subroutine particle_specifiers_2
```

14.4   Process library access

Processes (the code and data that are necessary for evaluating matrix elements of a particular process or process component) are organized in process libraries. In full form, process libraries contain generated and dynamically compiled and linked code, so they are actual libraries on the OS level. Alternatively, there may be simple processes that can be generated without referring to external libraries, and external libraries that are just linked in.

This module interfaces the OS to create, build, and use process libraries.

We work with two related data structures. There is the list of process configurations that stores the user input and data derived from it. A given process

configuration list is scanned for creating a process library, which consists of both data and code. The creation step involves calling external programs and incorporating external code.

For the subsequent integration and event generation steps, we read the process library. We also support partial (re)creation of the process library. To this end, we should be able to reconstruct the configuration data records from the process library.

⟨process_libraries.f90⟩≡
  ⟨*File header*⟩

  module process_libraries

    use iso_c_binding !NODEP!
    use kinds !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use md5
    use unit_tests
    use os_interface
    use lexers
    use variables
    use user_code_interface
    use models
    use flavors

    use process_constants
    use prclib_interfaces
    use prc_core_def
    use particle_specifiers

  ⟨*Standard module head*⟩

  ⟨*Process libraries: public*⟩

  ⟨*Process libraries: parameters*⟩

  ⟨*Process libraries: types*⟩

  ⟨*Process libraries: interfaces*⟩

  ⟨*Process libraries: test declarations*⟩

  contains

  ⟨*Process libraries: procedures*⟩

  ⟨*Process libraries: tests*⟩

  end module process_libraries

14.4.1 Auxiliary stuff

Here is a small subroutine that strips the left-hand side and the equals sign off an equation.

⟨*Process libraries: procedures*⟩≡
```
subroutine strip_prefix (buffer)
  character(*), intent(inout) :: buffer
  type(string_t) :: string, prefix
  string = buffer
  call split (string, prefix, "=")
  buffer = string
end subroutine strip_prefix
```

14.4.2 Process definition objects

We collect process configuration data in a derived type, `process_def_t`. A process can be a collection of several components which are treated as a single entity for the purpose of observables and event generation. Multiple process components may initially be defined by the user. The system may add additional components, e.g., subtraction terms. The common data type is `process_component_def_t`. Within each component, there are several universal data items, and a part which depend on the particular process variant. The latter is covered by an abstract type `prc_core_def_t` and its extensions.

Wrapper for components

We define a wrapper type for the configuration of individual components.

The string `basename` is used for building file, module, and function names for the current process component. Initially, it will be built from the corresponding process basename by appending an alphanumeric suffix.

The logical `initial` tells whether this is a user-defined (true) or system-generated (false) configuration.

The numbers `n_in`, `n_out`, and `n_tot` denote the incoming, outgoing and total number of particles (partons) participating in the process component, respectively. These are the nominal particles, as input by the user (recombination may change the particle content, for the output events).

The string arrays `prt_in` and `prt_out` hold the particle specifications as provided by the user. For a system-generated process component, they remain deallocated.

The `method` string is used to determine the type of process matrix element and how it is obtained.

The `description` string collects the information about particle content and method in a single human-readable string.

The pointer object `core_def` is allocated according to the actual process variant, which depends on the method. The subobject holds any additional configuration data that is relevant for the process component.

We assume that no finalizer is needed.

⟨*Process libraries: public*⟩≡
```
public :: process_component_def_t
```

⟨*Process libraries: types*⟩≡

```
type :: process_component_def_t
   private
   type(string_t) :: basename
   logical :: initial = .false.
   integer :: n_in = 0
   integer :: n_out = 0
   integer :: n_tot = 0
   type(prt_spec_t), dimension(:), allocatable :: prt_in
   type(prt_spec_t), dimension(:), allocatable :: prt_out
   type(string_t) :: method
   type(string_t) :: description
   class(prc_core_def_t), allocatable :: core_def
   character(32) :: md5sum = ""
 contains
 ⟨Process libraries: process component def: TBP⟩
end type process_component_def_t
```

Display the complete content.

⟨*Process libraries: process component def: TBP*⟩≡

```
procedure :: write => process_component_def_write
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_write (object, unit)
  class(process_component_def_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A,A)")  "Component ID        = ", char (object%basename)
  write (u, "(3x,A,L1)") "Initial component   = ", object%initial
  write (u, "(3x,A,I0,1x,I0,1x,I0)") "N (in, out, tot)    = ", &
       object%n_in, object%n_out, object%n_tot
  write (u, "(3x,A)", advance="no") "Particle content    = "
  if (allocated (object%prt_in)) then
     call prt_spec_write (object%prt_in, u, advance="no")
  else
     write (u, "(A)", advance="no")  "[undefined]"
  end if
  write (u, "(A)", advance="no") " => "
  if (allocated (object%prt_out)) then
     call prt_spec_write (object%prt_out, u, advance="no")
  else
     write (u, "(A)", advance="no")  "[undefined]"
  end if
  write (u, "(A)")
  if (object%method /= "") then
     write (u, "(3x,A,A)")  "Method              = ", &
          char (object%method)
  else
     write (u, "(3x,A)")  "Method              = [undefined]"
  end if
  if (allocated (object%core_def)) then
     write (u, "(3x,A,A)")  "Process variant     = ", &
          char (object%core_def%type_string ())
```

1832

```
    call object%core_def%write (u)
  else
     write (u, "(3x,A)")  "Process variant    = [undefined]"
  end if
  write (u, "(3x,A,A,A)") "MD5 sum (def)       = '", object%md5sum, "'"
end subroutine process_component_def_write
```

Read the process component definition. Allocate the process variant definition with appropriate type, matching the type name on file with the provided templates.

⟨*Process libraries: process component def: TBP*⟩+≡
```
    procedure :: read => process_component_def_read
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_component_def_read (component, unit, core_def_templates)
      class(process_component_def_t), intent(out) :: component
      integer, intent(in) :: unit
      type(prc_template_t), dimension(:), intent(in) :: core_def_templates
      character(80) :: buffer
      type(string_t) :: var_buffer, prefix, in_state, out_state
      type(string_t) :: variant_type

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      component%basename = trim (adjustl (buffer))

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      read (buffer, *)  component%initial

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      read (buffer, *)  component%n_in, component%n_out, component%n_tot

      call get (unit, var_buffer)
      call split (var_buffer, prefix, "=")   ! keeps 'in => out'
      call split (var_buffer, prefix, "=")   ! actually: separator is '=>'

      in_state = prefix
      if (component%n_in > 0) then
         call prt_spec_read (component%prt_in, in_state)
      end if

      out_state = extract (var_buffer, 2)
      if (component%n_out > 0) then
         call prt_spec_read (component%prt_out, out_state)
      end if

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      component%method = trim (adjustl (buffer))
      if (component%method == "[undefined]") &
           component%method = ""
```

```
    read (unit, "(A)")  buffer
    call strip_prefix (buffer)
    variant_type = trim (adjustl (buffer))
    call allocate_core_def &
        (core_def_templates, variant_type, component%core_def)
    if (allocated (component%core_def)) then
        call component%core_def%read (unit)
    end if

    read (unit, "(A)")  buffer
    call strip_prefix (buffer)
    read (buffer(3:34), "(A32)")  component%md5sum

  end subroutine process_component_def_read
```

Short account.

⟨*Process libraries: process component def: TBP*⟩+≡
```
    procedure :: show => process_component_def_show
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_component_def_show (object, unit)
      class(process_component_def_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(6x,A)", advance="no")  char (object%basename)
      if (.not. object%initial) &
          write (u, "('*')", advance="no")
      write (u, "(':',1x)", advance="no")
      if (allocated (object%prt_in)) then
         call prt_spec_write (object%prt_in, u, advance="no")
      else
         write (u, "(A)", advance="no")  "[undefined]"
      end if
      write (u, "(A)", advance="no") " => "
      if (allocated (object%prt_out)) then
         call prt_spec_write (object%prt_out, u, advance="no")
      else
         write (u, "(A)", advance="no")  "[undefined]"
      end if
      if (object%method /= "") then
         write (u, "(2x,'[',A,']')")  char (object%method)
      else
         write (u, *)
      end if
    end subroutine process_component_def_show
```

Compute the MD5 sum of a process component. We reset the stored MD5 sum
to the empty string (so a previous value is not included in the calculation), the
write a temporary file and calculate the MD5 sum of that file.

This implies that all data that are displayed by the `write` method become
part of the MD5 sum calculation.

The `model` is not part of the object, but must be included in the MD5 sum. Otherwise, modifying the model and nothing else would not trigger remaking the process-component source. Note that the model parameters may change later and therefore are not incorporated.

After the MD5 sum of the component has been computed, we communicate it to the `writer` subobject of the specific `core_def` component. Although these types are abstract, the MD5-related features are valid for the abstract types.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: compute_md5sum => process_component_def_compute_md5sum
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_component_def_compute_md5sum (component, model)
  class(process_component_def_t), intent(inout) :: component
  type(model_t), intent(in), pointer :: model
  integer :: u
  component%md5sum = ""
  u = free_unit ()
  open (u, status = "scratch", action = "readwrite")
  if (associated (model))  write (u, "(A32)")  model_get_md5sum (model)
  call component%write (u)
  rewind (u)
  component%md5sum = md5sum (u)
  close (u)
  if (allocated (component%core_def)) then
     call component%core_def%set_md5sum (component%md5sum)
  end if
end subroutine process_component_def_compute_md5sum
```

Allocate the process driver (with a suitable type) for a process component. For internal processes, we may set all data already at this stage.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: allocate_driver => process_component_def_allocate_driver
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_component_def_allocate_driver (component, driver)
  class(process_component_def_t), intent(in) :: component
  class(prc_core_driver_t), intent(out), allocatable :: driver
  if (allocated (component%core_def)) then
     call component%core_def%allocate_driver (driver, component%basename)
  end if
end subroutine process_component_def_allocate_driver
```

Tell whether the process core needs external code.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: needs_code => process_component_def_needs_code
```

⟨*Process libraries: procedures*⟩+≡
```
function process_component_def_needs_code (component) result (flag)
  class(process_component_def_t), intent(in) :: component
  logical :: flag
  flag = component%core_def%needs_code ()
end function process_component_def_needs_code
```

If there is external code, the `core_def` subobject should provide a writer object. This method returns a pointer to the writer.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: get_writer_ptr => process_component_def_get_writer_ptr
```

⟨*Process libraries: procedures*⟩+≡
```
function process_component_def_get_writer_ptr (component) result (writer)
  class(process_component_def_t), intent(in), target :: component
  class(prc_writer_t), pointer :: writer
  writer => component%core_def%writer
end function process_component_def_get_writer_ptr
```

Return an array which holds the names of all C functions that this process component implements.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: get_features => process_component_def_get_features
```

⟨*Process libraries: procedures*⟩+≡
```
function process_component_def_get_features (component) result (features)
  class(process_component_def_t), intent(in) :: component
  type(string_t), dimension(:), allocatable :: features
  call component%core_def%get_features (features)
end function process_component_def_get_features
```

Assign procedure pointers in the `driver` component (external processes). For internal processes, this is meaningless.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: connect => process_component_def_connect
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_component_def_connect &
    (component, lib_driver, i, proc_driver)
  class(process_component_def_t), intent(in) :: component
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  select type (proc_driver)
  class is (process_driver_internal_t)
     ! nothing to do
  class default
     call component%core_def%connect (lib_driver, i, proc_driver)
  end select
end subroutine process_component_def_connect
```

Return a pointer to the process core definition, which is of abstract type.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: get_core_def_ptr => process_component_get_core_def_ptr
```

⟨*Process libraries: procedures*⟩+≡
```
function process_component_get_core_def_ptr (component) result (ptr)
  class(process_component_def_t), intent(in), target :: component
  class(prc_core_def_t), pointer :: ptr
  ptr => component%core_def
end function process_component_get_core_def_ptr
```

Return nominal particle counts, as input by the user.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: get_n_in  => process_component_def_get_n_in
procedure :: get_n_out => process_component_def_get_n_out
procedure :: get_n_tot => process_component_def_get_n_tot
```

⟨*Process libraries: procedures*⟩+≡
```
function process_component_def_get_n_in (component) result (n_in)
  class(process_component_def_t), intent(in) :: component
  integer :: n_in
  n_in = component%n_in
end function process_component_def_get_n_in

function process_component_def_get_n_out (component) result (n_out)
  class(process_component_def_t), intent(in) :: component
  integer :: n_out
  n_out = component%n_out
end function process_component_def_get_n_out

function process_component_def_get_n_tot (component) result (n_tot)
  class(process_component_def_t), intent(in) :: component
  integer :: n_tot
  n_tot = component%n_tot
end function process_component_def_get_n_tot
```

Allocate and return string arrays for the incoming and outgoing particles.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: get_prt_in => process_component_def_get_prt_in
procedure :: get_prt_out => process_component_def_get_prt_out
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_component_def_get_prt_in (component, prt)
  class(process_component_def_t), intent(in) :: component
  type(string_t), dimension(:), intent(out), allocatable :: prt
  integer :: i
  allocate (prt (component%n_in))
  do i = 1, component%n_in
     prt(i) = component%prt_in(i)%to_string ()
  end do
end subroutine process_component_def_get_prt_in

subroutine process_component_def_get_prt_out (component, prt)
  class(process_component_def_t), intent(in) :: component
  type(string_t), dimension(:), intent(out), allocatable :: prt
  integer :: i
  allocate (prt (component%n_out))
  do i = 1, component%n_out
     prt(i) = component%prt_out(i)%to_string ()
  end do
end subroutine process_component_def_get_prt_out
```

Return the MD5 sum.

⟨*Process libraries: process component def: TBP*⟩+≡
```
procedure :: get_md5sum => process_component_def_get_md5sum
```

1837

⟨*Process libraries: procedures*⟩+≡
```
    function process_component_def_get_md5sum (component) result (md5sum)
      class(process_component_def_t), intent(in) :: component
      character(32) :: md5sum
      md5sum = component%md5sum
    end function process_component_def_get_md5sum
```

Process definition

The process component definitions are collected in a common process definition object.

The `id` is the ID string that the user has provided for identifying this process. It must be a string that is allowed as part of a Fortran variable name, since it may be used for generating code.

The number `n_in` is 1 or 2 for a decay or scattering process, respectively. This must be identical to `n_in` for all components.

The initial and extra component definitions (see above) are allocated as the `initial` and `extra` arrays, respectively. The latter are determined from the former.

The `md5sum` is used to verify the integrity of the configuration.

⟨*Process libraries: public*⟩+≡
```
    public :: process_def_t
```

⟨*Process libraries: types*⟩+≡
```
    type :: process_def_t
      private
      type(string_t) :: id
      integer :: num_id = 0
      type(model_t), pointer :: model => null ()
      type(string_t) :: model_name
      integer :: n_in  = 0
      integer :: n_initial = 0
      integer :: n_extra = 0
      type(process_component_def_t), dimension(:), allocatable :: initial
      type(process_component_def_t), dimension(:), allocatable :: extra
      character(32) :: md5sum = ""
     contains
     ⟨Process libraries: process def: TBP⟩
    end type process_def_t
```

Write the process definition including components:

⟨*Process libraries: process def: TBP*⟩≡
```
    procedure :: write => process_def_write
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_def_write (object, unit)
      class(process_def_t), intent(in) :: object
      integer, intent(in) :: unit
      integer :: i
      write (unit, "(1x,A,A,A)") "ID = '", char (object%id), "'"
      if (object%num_id /= 0) &
           write (unit, "(1x,A,I0)")  "ID(num) = ", object%num_id
```

```
      select case (object%n_in)
      case (1);  write (unit, "(1x,A)")  "Decay"
      case (2);  write (unit, "(1x,A)")  "Scattering"
      case default
         write (unit, "(1x,A)")  "[Undefined process]"
         return
      end select
      if (object%model_name /= "") then
         write (unit, "(1x,A,A)")  "Model = ", char (object%model_name)
      else
         write (unit, "(1x,A)")  "Model = [undefined]"
      end if
      write (unit, "(1x,A,I0)")  "Initially defined component(s) = ", &
           object%n_initial
      write (unit, "(1x,A,I0)")  "Extra generated component(s)   = ", &
           object%n_extra
      write (unit, "(1x,A,A,A)") "MD5 sum   = '", object%md5sum, "'"
      if (allocated (object%initial)) then
         do i = 1, size (object%initial)
            write (unit, "(1x,A,I0)")  "Component #", i
            call object%initial(i)%write (unit)
         end do
      end if
      if (allocated (object%extra)) then
         do i = 1, size (object%extra)
            write (unit, "(1x,A,I0)")  "Component #", object%n_initial + i
            call object%extra(i)%write (unit)
         end do
      end if
    end subroutine process_def_write
```

Read the process definition including components.

⟨*Process libraries: process def: TBP*⟩+≡
```
    procedure :: read => process_def_read
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_def_read (object, unit, core_def_templates)
      class(process_def_t), intent(out) :: object
      integer, intent(in) :: unit
      type(prc_template_t), dimension(:), intent(in) :: core_def_templates
      integer :: i, i1, i2
      character(80) :: buffer, ref
      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      i1 = scan (buffer, "'")
      i2 = scan (buffer, "'", back=.true.)
      if (i2 > i1) then
         object%id = buffer(i1+1:i2-1)
      else
         object%id = ""
      end if

      read (unit, "(A)")  buffer
      select case (buffer(2:11))
```

```fortran
      case ("Decay      "); object%n_in = 1
      case ("Scattering"); object%n_in = 2
      case default
         return
      end select

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      object%model_name = trim (adjustl (buffer))
      if (object%model_name == "[undefined]")  object%model_name = ""

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      read (buffer, *)  object%n_initial

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      read (buffer, *)  object%n_extra

      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      read (buffer(3:34), "(A32)")  object%md5sum

      if (object%n_initial > 0) then
         allocate (object%initial (object%n_initial))
         do i = 1, object%n_initial
            read (unit, "(A)")  buffer
            write (ref, "(1x,A,I0)")  "Component #", i
            if (buffer /= ref)  return                 ! Wrong component header
            call object%initial(i)%read (unit, core_def_templates)
         end do
      end if

   end subroutine process_def_read
```

Short account.

⟨*Process libraries: process def: TBP*⟩+≡
```fortran
      procedure :: show => process_def_show
```

⟨*Process libraries: procedures*⟩+≡
```fortran
   subroutine process_def_show (object, unit)
     class(process_def_t), intent(in) :: object
     integer, intent(in) :: unit
     integer :: i
     write (unit, "(4x,A)", advance="no") char (object%id)
     if (object%num_id /= 0) &
         write (unit, "(1x,'(',I0,')')", advance="no")  object%num_id
     if (object%model_name /= "") &
         write (unit, "(1x,'[',A,']')")  char (object%model_name)
     if (allocated (object%initial)) then
       do i = 1, size (object%initial)
          call object%initial(i)%show (unit)
       end do
     end if
```

```
      if (allocated (object%extra)) then
         do i = 1, size (object%extra)
            call object%extra(i)%show (unit)
         end do
      end if
    end subroutine process_def_show
```

Initialize an entry (initialize the process definition inside). We allocate the 'initial' set of components. Extra components remain unallocated.

The model should be present as a pointer. This allows us to retrieve the model's MD5 sum. However, for various tests it is sufficient to have the name.

We create the basenames for the process components by appending a suffix which we increment for each component.

⟨*Process libraries: process def: TBP*⟩+≡
```
    procedure :: init => process_def_init
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_def_init (def, id, &
         model, model_name, n_in, n_components, num_id)
      class(process_def_t), intent(out) :: def
      type(string_t), intent(in), optional :: id
      type(model_t), intent(in), optional, target :: model
      type(string_t), intent(in), optional :: model_name
      integer, intent(in), optional :: n_in
      integer, intent(in), optional :: n_components
      integer, intent(in), optional :: num_id
      character(16) :: suffix
      integer :: i
      if (present (id)) then
         def%id = id
      else
         def%id = ""
      end if
      if (present (num_id)) then
         def%num_id = num_id
      end if
      if (present (model)) then
         def%model => model
         def%model_name = model%get_name ()
      else
         def%model => null ()
         if (present (model_name)) then
            def%model_name = model_name
         else
            def%model_name = ""
         end if
      end if
      if (present (n_in))  def%n_in = n_in
      if (present (n_components)) then
         def%n_initial = n_components
         allocate (def%initial (n_components))
      end if
      def%initial%initial = .true.
      def%initial%method      = ""
```

```
      do i = 1, def%n_initial
         write (suffix, "(A,I0)")  "_i", i
         def%initial(i)%basename = def%id // trim (suffix)
      end do
      def%initial%description = ""
    end subroutine process_def_init
```

Initialize an initial component. The particle content must be specified. The process core block is not (yet) allocated.

    We assume that the particle arrays match the `n_in` and `n_out` values in size. The model is referred to by name; it is identified as an existing model later. The index `i` must refer to an existing element of the component array.

    Data specific for the process core of a component are imported as the `core_def` argument. We should allocate an object of class `prc_core_def_t` with the appropriate specific type, fill it, and transfer it to the process component definition here. The allocation is moved, so the original allocated object is returned empty.

⟨*Process libraries: process def: TBP*⟩+≡
```
    procedure :: import_component => process_def_import_component
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_def_import_component (def, &
        i, n_out, prt_in, prt_out, method, variant)
      class(process_def_t), intent(inout) :: def
      integer, intent(in) :: i
      integer, intent(in), optional :: n_out
      type(prt_spec_t), dimension(:), intent(in), optional :: prt_in
      type(prt_spec_t), dimension(:), intent(in), optional :: prt_out
      type(string_t), intent(in), optional :: method
      class(prc_core_def_t), &
          intent(inout), allocatable, optional :: variant
      integer :: p
      associate (comp => def%initial(i))
        if (present (n_out)) then
           comp%n_in  = def%n_in
           comp%n_out = n_out
           comp%n_tot = def%n_in + n_out
        end if
        if (present (prt_in)) then
           allocate (comp%prt_in (size (prt_in)))
           comp%prt_in = prt_in
        end if
        if (present (prt_out)) then
           allocate (comp%prt_out (size (prt_out)))
           comp%prt_out = prt_out
        end if
        if (present (method))  comp%method = method
        if (present (variant)) then
           call move_alloc (variant, comp%core_def)
        end if
        if (allocated (comp%prt_in) .and. allocated (comp%prt_out)) then
           associate (d => comp%description)
             d = ""
```

1842

```
          do p = 1, size (prt_in)
             if (p > 1)  d = d // ", "
             d = d // comp%prt_in(p)%to_string ()
          end do
          d = d // " => "
          do p = 1, size (prt_out)
             if (p > 1)  d = d // ", "
             d = d // comp%prt_out(p)%to_string ()
          end do
          if (comp%method /= "") then
             d = d // " [" // comp%method // "]"
          end if
       end associate
     end if
   end associate
 end subroutine process_def_import_component
```

Compute the MD5 sum for this process definition. We compute the MD5 sums for all components individually, than concatenate a string of those and compute the MD5 sum of this string. We also include the model name. All other data part of the component definitions.

⟨*Process libraries: process def: TBP*⟩+≡
```
    procedure :: compute_md5sum => process_def_compute_md5sum
```

⟨*Process libraries: procedures*⟩+≡
```
  subroutine process_def_compute_md5sum (def)
    class(process_def_t), intent(inout) :: def
    integer :: i
    type(string_t) :: buffer
    buffer = def%model_name
    do i = 1, def%n_initial
       call def%initial(i)%compute_md5sum (def%model)
       buffer = buffer // def%initial(i)%md5sum
    end do
    do i = 1, def%n_extra
       call def%extra(i)%compute_md5sum (def%model)
       buffer = buffer // def%initial(i)%md5sum
    end do
    def%md5sum = md5sum (char (buffer))
  end subroutine process_def_compute_md5sum
```

Return the MD5 sum of the process or of a process component.

⟨*Process libraries: process def: TBP*⟩+≡
```
    procedure :: get_md5sum => process_def_get_md5sum
```

⟨*Process libraries: procedures*⟩+≡
```
  function process_def_get_md5sum (def, i_component) result (md5sum)
    class(process_def_t), intent(in) :: def
    integer, intent(in), optional :: i_component
    character(32) :: md5sum
    if (present (i_component)) then
       md5sum = def%initial(i_component)%md5sum
    else
       md5sum = def%md5sum
```

```
      end if
    end function process_def_get_md5sum
```

This query tells whether a specific process component relies on external code. This includes all traditional WHIZARD matrix elements which rely on O'MEGA for code generation. Other process components (trivial decays, subtraction terms) do not require external code.

NOTE: Implemented only for initial component.

The query is passed to the process component.

⟨*Process libraries: process def: TBP*⟩+≡
```
    procedure :: needs_code => process_def_needs_code
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_def_needs_code (def, i_component) result (flag)
      class(process_def_t), intent(in) :: def
      integer, intent(in) :: i_component
      logical :: flag
      flag = def%initial(i_component)%needs_code ()
    end function process_def_needs_code
```

Process definition list

A list of process definitions is the starting point for creating a process library. The list is built when reading the user input. When reading an existing process library, the list is used for cross-checking and updating the configuration.

We need a type for the list entry. The simplest way is to extend the process definition type, so all methods apply to the process definition directly.

⟨*Process libraries: public*⟩+≡
```
    public :: process_def_entry_t
```

⟨*Process libraries: types*⟩+≡
```
    type, extends (process_def_t) :: process_def_entry_t
      private
      type(process_def_entry_t), pointer :: next => null ()
    end type process_def_entry_t
```

This is the type for the list itself.

⟨*Process libraries: types*⟩+≡
```
    type :: process_def_list_t
      private
      type(process_def_entry_t), pointer :: first => null ()
      type(process_def_entry_t), pointer :: last => null ()
     contains
     ⟨Process libraries: process def list: TBP⟩
    end type process_def_list_t
```

The deallocates the list iteratively. We assume that the list entries do not need finalization themselves.

⟨*Process libraries: process def list: TBP*⟩≡
```
    procedure :: final => process_def_list_final
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_final (list)
  class(process_def_list_t), intent(inout) :: list
  type(process_def_entry_t), pointer :: current
  nullify (list%last)
  do while (associated (list%first))
     current => list%first
     list%first => current%next
     deallocate (current)
  end do
end subroutine process_def_list_final
```

Write the complete list.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: write => process_def_list_write
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_write (object, unit, libpath)
  class(process_def_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: libpath
  type(process_def_entry_t), pointer :: entry
  integer :: i, u
  u = output_unit (unit)
  if (associated (object%first)) then
     i = 1
     entry => object%first
     do while (associated (entry))
        write (u, "(1x,A,I0,A)")  "Process #", i, ":"
        call entry%write (u)
        i = i + 1
        entry => entry%next
        if (associated (entry))  write (u, *)
     end do
  else
     write (u, "(1x,A)")  "Process definition list: [empty]"
  end if
end subroutine process_def_list_write
```

Short account.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: show => process_def_list_show
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_show (object, unit)
  class(process_def_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(process_def_entry_t), pointer :: entry
  integer :: u
  u = output_unit (unit)
  if (associated (object%first)) then
     write (u, "(2x,A)")  "Processes:"
     entry => object%first
     do while (associated (entry))
```

```
            call entry%show (u)
            entry => entry%next
         end do
      else
         write (u, "(2x,A)")  "Processes: [empty]"
      end if
   end subroutine process_def_list_show
```

Read the complete list. We need an array of templates for the component sub-objects of abstract `prc_core_t` type, to allocate them with the correct specific type.

NOTE: Error handling is missing. Reading will just be aborted on error, or an I/O error occurs.

⟨*Process libraries: process def list: TBP*⟩+≡
```
   procedure :: read => process_def_list_read
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_def_list_read (object, unit, core_def_templates)
     class(process_def_list_t), intent(out) :: object
     integer, intent(in) :: unit
     type(prc_template_t), dimension(:), intent(in) :: core_def_templates
     type(process_def_entry_t), pointer :: entry
     character(80) :: buffer, ref
     integer :: i
     read (unit, "(A)")  buffer
     write (ref, "(1x,A)")  "Process definition list: [empty]"
     if (buffer == ref)  return          ! OK: empty library
     backspace (unit)
     READ_ENTRIES: do i = 1, huge (0)
        if (i > 1) read (unit, *, end=1)
        read (unit, "(A)")  buffer

        write (ref, "(1x,A,I0,A)")  "Process #", i, ":"
        if (buffer /= ref)  return       ! Wrong process header: done.
        allocate (entry)
        call entry%read (unit, core_def_templates)
        call object%append (entry)
     end do READ_ENTRIES
 1   continue                            ! EOF: done
   end subroutine process_def_list_read
```

Append an entry to the list. The entry should be allocated as a pointer, and the pointer allocation is transferred. The original pointer is returned null.

⟨*Process libraries: process def list: TBP*⟩+≡
```
   procedure :: append => process_def_list_append
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_def_list_append (list, entry)
     class(process_def_list_t), intent(inout) :: list
     type(process_def_entry_t), intent(inout), pointer :: entry
     if (list%contains (entry%id)) then
        call msg_fatal ("Recording process: '" // char (entry%id) &
             // "' has already been defined")
```

```
      end if
      if (associated (list%first)) then
         list%last%next => entry
      else
         list%first => entry
      end if
      list%last => entry
      entry => null ()
   end subroutine process_def_list_append
```

Probe the process definition list

Return the number of processes supported by the library.

⟨*Process libraries: process def list: TBP*⟩+≡
```
   procedure :: get_n_processes => process_def_list_get_n_processes
```

⟨*Process libraries: procedures*⟩+≡
```
   function process_def_list_get_n_processes (list) result (n)
     integer :: n
     class(process_def_list_t), intent(in) :: list
     type(process_def_entry_t), pointer :: current
     n = 0
     current => list%first
     do while (associated (current))
        n = n + 1
        current => current%next
     end do
   end function process_def_list_get_n_processes
```

Allocate an array with the process IDs supported by the library.

⟨*Process libraries: process def list: TBP*⟩+≡
```
   procedure :: get_process_id_list => process_def_list_get_process_id_list
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_def_list_get_process_id_list (list, id)
     class(process_def_list_t), intent(in) :: list
     type(string_t), dimension(:), allocatable, intent(out) :: id
     type(process_def_entry_t), pointer :: current
     integer :: i
     allocate (id (list%get_n_processes ()))
     i = 0
     current => list%first
     do while (associated (current))
        i = i + 1
        id(i) = current%id
        current => current%next
     end do
   end subroutine process_def_list_get_process_id_list
```

Return true if a given process is in the library.

⟨*Process libraries: process def list: TBP*⟩+≡
```
   procedure :: contains => process_def_list_contains
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_def_list_contains (list, id) result (flag)
      logical :: flag
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      type(process_def_entry_t), pointer :: current
      current => list%first
      do while (associated (current))
         if (id == current%id) then
            flag = .true.;   return
         end if
         current => current%next
      end do
      flag = .false.
    end function process_def_list_contains
```

Return the index of the entry that corresponds to a given process.

⟨*Process libraries: process def list: TBP*⟩+≡
```
    procedure :: get_entry_index => process_def_list_get_entry_index
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_def_list_get_entry_index (list, id) result (n)
      integer :: n
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      type(process_def_entry_t), pointer :: current
      n = 0
      current => list%first
      do while (associated (current))
         n = n + 1
         if (id == current%id) then
            return
         end if
         current => current%next
      end do
      n = 0
    end function process_def_list_get_entry_index
```

Return the numerical ID for a process.

⟨*Process libraries: process def list: TBP*⟩+≡
```
    procedure :: get_num_id => process_def_list_get_num_id
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_def_list_get_num_id (list, id) result (num_id)
      integer :: num_id
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      type(process_def_entry_t), pointer :: current
      current => list%first
      do while (associated (current))
         if (id == current%id) then
            num_id = current%num_id
            return
         end if
```

```
      current => current%next
    end do
    num_id = 0
  end function process_def_list_get_num_id
```

Return the model name for a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
    procedure :: get_model_name => process_def_list_get_model_name
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_def_list_get_model_name (list, id) result (model_name)
    type(string_t) :: model_name
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    type(process_def_entry_t), pointer :: current
    current => list%first
    do while (associated (current))
      if (id == current%id) then
        model_name = current%model_name
        return
      end if
      current => current%next
    end do
    model_name = ""
  end function process_def_list_get_model_name
```

Return the number of incoming particles of a given process in the library. This tells us whether the process is a decay or a scattering.

⟨*Process libraries: process def list: TBP*⟩+≡

```
    procedure :: get_n_in => process_def_list_get_n_in
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_def_list_get_n_in (list, id) result (n)
    integer :: n
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    type(process_def_entry_t), pointer :: current
    current => list%first
    do while (associated (current))
      if (id == current%id) then
        n = current%n_in
        return
      end if
      current => current%next
    end do
  end function process_def_list_get_n_in
```

Return the number of components of a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
    procedure :: get_n_components => process_def_list_get_n_components
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_def_list_get_n_components (list, id) result (n)
```

```
      integer :: n
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      type(process_def_entry_t), pointer :: current
      current => list%first
      do while (associated (current))
         if (id == current%id) then
            n = current%n_initial + current%n_extra
            return
         end if
         current => current%next
      end do
    end function process_def_list_get_n_components
```

Return a pointer to a specific process component definition.

⟨*Process libraries: process def list: TBP*⟩+≡

```
    procedure :: get_component_def_ptr => process_def_list_get_component_def_ptr
```

⟨*Process libraries: procedures*⟩+≡

```
    function process_def_list_get_component_def_ptr (list, id, i) result (ptr)
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      integer, intent(in) :: i
      type(process_component_def_t), pointer :: ptr
      type(process_def_entry_t), pointer :: current
      ptr => null ()
      current => list%first
      do while (associated (current))
         if (id == current%id) then
            if (i <= current%n_initial) then
               ptr => current%initial(i)
            else if (i <= current%n_initial + current%n_extra) then
               ptr => current%extra(i-current%n_initial)
            end if
            return
         end if
         current => current%next
      end do
    end function process_def_list_get_component_def_ptr
```

Return the list of component IDs of a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
    procedure :: get_component_list => process_def_list_get_component_list
```

⟨*Process libraries: procedures*⟩+≡

```
    subroutine process_def_list_get_component_list (list, id, cid)
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      type(string_t), dimension(:), allocatable, intent(out) :: cid
      type(process_def_entry_t), pointer :: current
      integer :: i, n
      current => list%first
      do while (associated (current))
         if (id == current%id) then
```

```
            allocate (cid (current%n_initial + current%n_extra))
            do i = 1, current%n_initial
               cid(i) = current%initial(i)%basename
            end do
            n = current%n_initial
            do i = 1, current%n_extra
               cid(n + i) = current%extra(i)%basename
            end do
            return
         end if
         current => current%next
      end do
   end subroutine process_def_list_get_component_list
```

Return the list of component description strings for a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡
```
    procedure :: get_component_description_list => &
        process_def_list_get_component_description_list
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_def_list_get_component_description_list &
        (list, id, description)
      class(process_def_list_t), intent(in) :: list
      type(string_t), intent(in) :: id
      type(string_t), dimension(:), allocatable, intent(out) :: description
      type(process_def_entry_t), pointer :: current
      integer :: i, n
      current => list%first
      do while (associated (current))
         if (id == current%id) then
            allocate (description (current%n_initial + current%n_extra))
            do i = 1, current%n_initial
               description(i) = current%initial(i)%description
            end do
            n = current%n_initial
            do i = 1, current%n_extra
               description(n + i) = current%extra(i)%description
            end do
            return
         end if
         current => current%next
      end do
   end subroutine process_def_list_get_component_description_list
```

14.4.3   Process library

The process library object is the interface between the process definition data, as provided by the user, generated or linked process code on file, and the process run data that reference the process code.

Process library entry

For each process component that is part of the library, there is a separate library entry (`process_library_entry_t`. The library entry connects a process definition with the specific code (if any) in the compiled driver library.

The `status` indicates how far the process has been processed by the system (definition, code generation, compilation, linking). A process with status `STAT_LOADED` is accessible for computing matrix elements.

The `def` pointer identifies the corresponding process definition. The process component within that definition is identified by the `i_component` index.

The `i_external` index refers to the compiled library driver. If it is zero, there is no associated matrix-element code.

The `driver` component holds the pointers to the matrix-element specific functions, in particular the matrix element function itself.

⟨*Process libraries: types*⟩+≡
```
type :: process_library_entry_t
  private
  integer :: status = STAT_UNKNOWN
  type(process_def_t), pointer :: def => null ()
  integer :: i_component = 0
  integer :: i_external = 0
  class(prc_core_driver_t), allocatable :: driver
 contains
 ⟨Process libraries: process library entry: TBP⟩
end type process_library_entry_t
```

Here are the available status codes. An entry starts with `UNKNOWN` status. Once the association with a valid process definition is established, the status becomes `CONFIGURED`. If matrix element source code is to be generated by the system or provided from elsewhere, `CODE_GENERATED` indicates that this is done. The `COMPILED` status is next, it also applies to processes which are accessed as precompiled binaries. Finally, the library is linked and process pointers are set; this is marked as `LOADED`.

For a process library, the initial status is `OPEN`, since process definitions may be added. After configuration, the process content is fixed and the status becomes `CONFIGURED`. The further states are as above, always referring to the lowest status among the process entries.

⟨*Process libraries: parameters*⟩≡
```
integer, parameter :: STAT_UNKNOWN = 0
integer, parameter :: STAT_OPEN = 1
integer, parameter :: STAT_CONFIGURED = 2
integer, parameter :: STAT_SOURCE = 3
integer, parameter :: STAT_COMPILED = 4
integer, parameter :: STAT_LINKED = 5
integer, parameter :: STAT_ACTIVE = 6
```

These are the associated code letters, for output:

⟨*Process libraries: parameters*⟩+≡
```
character, dimension(0:6), parameter :: STATUS_LETTER = &
     ["?", "o", "f", "s", "c", "l", "a"]
```

1852

This produces a condensed account of the library entry. The status is indicated by a letter in brackets, then the ID and component index of the associated process definition, finally the library index, if available.

⟨*Process libraries: process library entry: TBP*⟩≡
```
    procedure :: to_string => process_library_entry_to_string
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_entry_to_string (object) result (string)
      type(string_t) :: string
      class(process_library_entry_t), intent(in) :: object
      character(32) :: buffer
      string = "[" // STATUS_LETTER(object%status) // "]"
      select case (object%status)
      case (STAT_UNKNOWN)
      case default
         if (associated (object%def)) then
            write (buffer, "(I0)")  object%i_component
            string = string // " " // object%def%id // "." // trim (buffer)
         end if
         if (object%i_external /= 0) then
            write (buffer, "(I0)")  object%i_external
            string = string // " = ext:" // trim (buffer)
         else
            string = string // " = int"
         end if
         if (allocated (object%driver)) then
            string = string // " (" // object%driver%type_name () // ")"
         end if
      end select
    end function process_library_entry_to_string
```

Initialize with data. Used for the unit tests.

⟨*Process libraries: process library entry: TBP*⟩+≡
```
    procedure :: init => process_library_entry_init
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_entry_init (object, &
         status, def, i_component, i_external)
      class(process_library_entry_t), intent(out) :: object
      integer, intent(in) :: status
      type(process_def_t), target, intent(in) :: def
      integer, intent(in) :: i_component
      integer, intent(in) :: i_external
      object%status = status
      object%def => def
      object%i_component = i_component
      object%i_external = i_external
    end subroutine process_library_entry_init
```

Assign pointers for all process-specific features. We have to combine the method from the `core_def` specification, the assigned pointers within the library driver, the index within that driver, and the process driver which should receive the links.

⟨*Process libraries: process library entry: TBP*⟩+≡

```
        procedure :: connect => process_library_entry_connect
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_entry_connect (entry, lib_driver, i)
      class(process_library_entry_t), intent(inout) :: entry
      class(prclib_driver_t), intent(in) :: lib_driver
      integer, intent(in) :: i
      call entry%def%initial(entry%i_component)%connect &
           (lib_driver, i, entry%driver)
    end subroutine process_library_entry_connect
```


The process library object

The `process_library_t` type is an extension of the `process_def_list_t` type. Thus, it automatically contains the process definition list.

The `basename` identifies the library generically.

The `external` flag is true if any process within the library needs external code, so the library must correspond to an actual code library (statically or dynamically linked).

The `entry` array contains all process components that can be handled by this library. Each entry refers to the process (component) definition and to the associated external matrix element code, if there is any.

The `driver` object is needed only if `external` is true. This object handles all interactions with external matrix-element code.

The `md5sum` summarizes the complete `process_def_list_t` base object. It can be used to check if the library configuration has changed.

⟨*Process libraries: public*⟩+≡
```
    public :: process_library_t
```

⟨*Process libraries: types*⟩+≡
```
    type, extends (process_def_list_t) :: process_library_t
      private
      type(string_t) :: basename
      integer :: n_entries = 0
      logical :: external = .false.
      integer :: status = STAT_UNKNOWN
      logical :: static = .false.
      logical :: driver_exists = .false.
      logical :: makefile_exists = .false.
      type(process_library_entry_t), dimension(:), allocatable :: entry
      class(prclib_driver_t), allocatable :: driver
      character(32) :: md5sum = ""
    contains
    ⟨Process libraries: process library: TBP⟩
    end type process_library_t
```

For the output, we write first the metadata and the DL access record, then the library entries in short form, and finally the process definition list which is the base object.

Don't write the MD5 sum since this is used to generate it.

⟨*Process libraries: process library: TBP*⟩≡
```
    procedure :: write => process_library_write
```

1854

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_write (object, unit, libpath)
  class(process_library_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: libpath
  integer :: i, u
  u = output_unit (unit)
  write (u, "(1x,A,A)")  "Process library: ", char (object%basename)
  write (u, "(3x,A,L1)")   "external       = ", object%external
  write (u, "(3x,A,L1)")   "makefile exists = ", object%makefile_exists
  write (u, "(3x,A,L1)")   "driver exists  = ", object%driver_exists
  write (u, "(3x,A,A1)")   "code status    = ", &
       STATUS_LETTER (object%status)
  write (u, *)
  if (allocated (object%entry)) then
     write (u, "(1x,A)", advance="no")  "Process library entries:"
     write (u, "(1x,I0)")  object%n_entries
     do i = 1, size (object%entry)
        write (u, "(1x,A,I0,A,A)")  "Entry #", i, ": ", &
             char (object%entry(i)%to_string ())
     end do
     write (u, *)
  end if
  if (object%external) then
     call object%driver%write (u, libpath)
     write (u, *)
  end if
  call object%process_def_list_t%write (u)
end subroutine process_library_write
```

Condensed version for screen output.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: show => process_library_show
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_show (object, unit)
  class(process_library_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(A,A)")  "Process library: ", char (object%basename)
  write (u, "(2x,A,L1)")   "external       = ", object%external
  if (object%static) then
     write (u, "(2x,A,L1)")   "static         = ", .true.
  else
     write (u, "(2x,A,L1)")   "makefile exists = ", object%makefile_exists
     write (u, "(2x,A,L1)")   "driver exists  = ", object%driver_exists
  end if
  write (u, "(2x,A,A1)", advance="no")   "code status    = "
  select case (object%status)
  case (STAT_UNKNOWN);    write (u, "(A)")  "[unknown]"
  case (STAT_OPEN);       write (u, "(A)")  "open"
  case (STAT_CONFIGURED); write (u, "(A)")  "configured"
  case (STAT_SOURCE);     write (u, "(A)")  "source code exists"
```

```
      case (STAT_COMPILED);   write (u, "(A)")   "compiled"
      case (STAT_LINKED);     write (u, "(A)")   "linked"
      case (STAT_ACTIVE);     write (u, "(A)")   "active"
      end select
      call object%process_def_list_t%show (u)
   end subroutine process_library_show
```

The initializer defines just the basename. We may now add process definitions to the library.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: init => process_library_init
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_init (lib, basename)
      class(process_library_t), intent(out) :: lib
      type(string_t), intent(in) :: basename
      lib%basename = basename
      lib%status = STAT_OPEN
      call msg_message ("Process library '" // char (basename) &
            // "': initialized")
    end subroutine process_library_init
```

This alternative initializer declares the library as static. We should now add process definitions to the library, but all external process code exists already. We need the driver object, and we should check the defined processes against the stored ones.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: init_static => process_library_init_static
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_init_static (lib, basename)
      class(process_library_t), intent(out) :: lib
      type(string_t), intent(in) :: basename
      lib%basename = basename
      lib%status = STAT_OPEN
      lib%static = .true.
      call msg_message ("Static process library '" // char (basename) &
            // "': initialized")
    end subroutine process_library_init_static
```

The `configure` procedure scans the allocated entries in the process definition list. The configuration proceeds in three passes.

In the first pass, we scan the process definition list and count the number of process components and the number of components which need external code. This is used to allocate the `entry` array.

In the second pass, we initialize the `entry` elements which connect process definitions, process driver objects, and external code.

In the third pass, we initialize the library driver object, allocating an entry for each external matrix element.

NOTE: Currently we handle only `initial` process components; `extra` components are ignored.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: configure => process_library_configure
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_configure (lib, os_data)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  type(process_def_entry_t), pointer :: def_entry
  integer :: n_entries, n_external, i_entry, i_external
  type(string_t) :: model_name
  integer :: i_component

  n_entries = 0
  n_external = 0
  if (allocated (lib%entry))  deallocate (lib%entry)

  def_entry => lib%first
  do while (associated (def_entry))
     do i_component = 1, def_entry%n_initial
        n_entries = n_entries + 1
        if (def_entry%initial(i_component)%needs_code ()) then
           n_external = n_external + 1
           lib%external = .true.
        end if
     end do
     def_entry => def_entry%next
  end do
  lib%n_entries = n_entries

  allocate (lib%entry (n_entries))
  i_entry = 0
  i_external = 0
  def_entry => lib%first
  do while (associated (def_entry))
     do i_component = 1, def_entry%n_initial
        i_entry = i_entry + 1
        associate (lib_entry => lib%entry(i_entry))
          lib_entry%status = STAT_CONFIGURED
          lib_entry%def => def_entry%process_def_t
          lib_entry%i_component = i_component
          if (def_entry%initial(i_component)%needs_code ()) then
             i_external = i_external + 1
             lib_entry%i_external = i_external
          end if
          call def_entry%initial(i_component)%allocate_driver &
               (lib_entry%driver)
        end associate
     end do
     def_entry => def_entry%next
  end do

  call dispatch_prclib_driver (lib%driver, &
       lib%basename, lib%get_modellibs_ldflags (os_data))
  call lib%driver%init (n_external)
  do i_entry = 1, n_entries
     associate (lib_entry => lib%entry(i_entry))
       i_component = lib_entry%i_component
```

```
        model_name = lib_entry%def%model_name
        associate (def => lib_entry%def%initial(i_component))
          if (def%needs_code ()) then
             call lib%driver%set_record (lib_entry%i_external, &
                  def%basename, &
                  model_name, &
                  def%get_features (), def%get_writer_ptr ())
          end if
        end associate
      end associate
    end do

    if (lib%static) then
       if (lib%n_entries /= 0)  lib%entry%status = STAT_LINKED
       lib%status = STAT_LINKED
    else if (lib%external) then
       where (lib%entry%i_external == 0)  lib%entry%status = STAT_LINKED
       lib%status = STAT_CONFIGURED
       lib%makefile_exists = .false.
       lib%driver_exists = .false.
    else
       if (lib%n_entries /= 0)  lib%entry%status = STAT_LINKED
       lib%status = STAT_LINKED
    end if
  end subroutine process_library_configure
```

Compute the MD5 sum. We concatenate the individual MD5 sums of all processes (which, in turn, are derived from the MD5 sums of their components) and compute the MD5 sum of that.

This should be executed *after* configuration, where the driver was initialized, since otherwise the MD5 sum stored in the driver would be overwritten.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: compute_md5sum => process_library_compute_md5sum
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_compute_md5sum (lib)
      class(process_library_t), intent(inout) :: lib
      type(process_def_entry_t), pointer :: def_entry
      type(string_t) :: buffer
      buffer = lib%basename
      def_entry => lib%first
      do while (associated (def_entry))
         call def_entry%compute_md5sum ()
         buffer = buffer // def_entry%md5sum
         def_entry => def_entry%next
      end do
      lib%md5sum = md5sum (char (buffer))
      call lib%driver%set_md5sum (lib%md5sum)
    end subroutine process_library_compute_md5sum
```

Write an appropriate makefile, if there are external processes. Unless `force` is in effect, first check if there is already a makefile with the correct MD5 sum. If

yes, do nothing.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: write_makefile => process_library_write_makefile
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_write_makefile (lib, os_data, force, testflag)
      class(process_library_t), intent(inout) :: lib
      type(os_data_t), intent(in) :: os_data
      logical, intent(in) :: force
      logical, intent(in), optional :: testflag
      character(32) :: md5sum_file
      logical :: generate
      integer :: unit
      if (lib%external .and. .not. lib%static) then
         generate = .true.
         if (.not. force) then
            md5sum_file = lib%driver%get_md5sum_makefile ()
            if (lib%md5sum == md5sum_file) then
               call msg_message ("Process library '" // char (lib%basename) &
                    // "': keeping makefile")
               generate = .false.
            end if
         end if
         if (generate) then
            call msg_message ("Process library '" // char (lib%basename) &
                 // "': writing makefile")
            unit = free_unit ()
            open (unit, file = char (lib%driver%basename // ".makefile"), &
                 status="replace", action="write")
            call lib%driver%generate_makefile (unit, os_data, testflag)
            close (unit)
         end if
         lib%makefile_exists = .true.
      end if
    end subroutine process_library_write_makefile
```

Write the driver source code for the library to file, if there are external processes.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: write_driver => process_library_write_driver
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_write_driver (lib, force)
      class(process_library_t), intent(inout) :: lib
      logical, intent(in) :: force
      character(32) :: md5sum_file
      logical :: generate
      integer :: unit
      if (lib%external .and. .not. lib%static) then
         generate = .true.
         if (.not. force) then
            md5sum_file = lib%driver%get_md5sum_driver ()
            if (lib%md5sum == md5sum_file) then
               call msg_message ("Process library '" // char (lib%basename) &
                    // "': keeping driver")
```

```
            generate = .false.
         end if
      end if
      if (generate) then
         call msg_message ("Process library '" // char (lib%basename) &
               // "': writing driver")
         unit = free_unit ()
         open (unit, file = char (lib%driver%basename // ".f90"), &
               status="replace", action="write")
         call lib%driver%generate_driver_code (unit)
         close (unit)
      end if
      lib%driver_exists = .true.
   end if
 end subroutine process_library_write_driver
```

Update the compilation status of an external library.

Strictly speaking, this is not necessary for a one-time run, since the individual library methods will update the status themselves. However, it allows us to identify compilation steps that we can skip because the file exists or is already loaded, for the whole library or for particular entries.

Independently, the building process is controlled by a makefile. Thus, previous files are reused if they are not modified by the current compilation.

1. If it is not already loaded, attempt to load the library. If successful, check the overall MD5 sum. If it matches, just keep it loaded and mark as ACTIVE. If not, check the MD5 sum for all linked process components. Where it matches, mark the entry as COMPILED. Then, unload the library and mark as CONFIGURED.

   Thus, we can identify compiled files for all matrix elements which are accessible via the previous compiled library, even if it is no longer up to date.

2. If the library is now in CONFIGURED state, look for valid source files. Each entry that is just in CONFIGURED state will advance to SOURCE if the MD5 sum matches. Finally, advance the whole library to SOURCE if all entries are at least in this condition.

⟨*Process libraries: process library: TBP*⟩+≡
```
   procedure :: update_status => process_library_update_status
```

⟨*Process libraries: procedures*⟩+≡
```
   subroutine process_library_update_status (lib, os_data)
     class(process_library_t), intent(inout) :: lib
     type(os_data_t), intent(in) :: os_data
     character(32) :: md5sum_file
     integer :: i, i_external, i_component
     if (lib%external) then
        select case (lib%status)
        case (STAT_CONFIGURED:STAT_LINKED)
           call lib%driver%load (os_data, noerror=.true.)
        end select
        if (lib%driver%loaded) then
```

1860

```
            md5sum_file = lib%driver%get_md5sum (0)
            if (lib%md5sum == md5sum_file) then
                call lib%load_entries ()
                lib%entry%status = STAT_ACTIVE
                lib%status = STAT_ACTIVE
                call msg_message ("Process library '" // char (lib%basename) &
                     // "': active")
            else
                do i = 1, lib%n_entries
                    associate (entry => lib%entry(i))
                      i_external = entry%i_external
                      i_component = entry%i_component
                      if (i_external /= 0) then
                         md5sum_file = lib%driver%get_md5sum (i_external)
                         if (entry%def%get_md5sum (i_component) == md5sum_file) then
                            entry%status = STAT_COMPILED
                         else
                            entry%status = STAT_CONFIGURED
                         end if
                      end if
                    end associate
                end do
                call lib%driver%unload ()
                lib%status = STAT_CONFIGURED
            end if
        end if
        select case (lib%status)
        case (STAT_CONFIGURED)
            do i = 1, lib%n_entries
                associate (entry => lib%entry(i))
                  i_external = entry%i_external
                  i_component = entry%i_component
                  if (i_external /= 0) then
                     select case (entry%status)
                     case (STAT_CONFIGURED)
                        md5sum_file = lib%driver%get_md5sum_source (i_external)
                        if (entry%def%get_md5sum (i_component) == md5sum_file) then
                           entry%status = STAT_SOURCE
                        end if
                     end select
                  end if
                end associate
            end do
            if (all (lib%entry%status >= STAT_SOURCE)) then
                md5sum_file = lib%driver%get_md5sum_driver ()
                if (lib%md5sum == md5sum_file) then
                    lib%status = STAT_SOURCE
                end if
            end if
        end select
    end if
  end subroutine process_library_update_status
```

This procedure triggers code generation for all processes where this is possible.

We generate code only for external processes of status `STAT_CONFIGURED`, which then advance to `STAT_SOURCE`. If, for a particular process, the status is already advanced, we do not remove previous files, so `make` will consider them as up to date if they exist. Otherwise, we remove those files to force a fresh `make`.

Finally, if any source code has been generated, we need a driver file.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: make_source => process_library_make_source
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_make_source (lib, os_data, keep_old_source)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: keep_old_source
  logical :: keep_old
  integer :: i, i_external
  keep_old = .false.
  if (present (keep_old_source))  keep_old = keep_old_source
  if (lib%external .and. .not. lib%static) then
     select case (lib%status)
     case (STAT_CONFIGURED)
        if (keep_old) then
           call msg_message ("Process library '" // char (lib%basename) &
                // "': keeping source code")
        else
           call msg_message ("Process library '" // char (lib%basename) &
                // "': creating source code")
           do i = 1, size (lib%entry)
              associate (entry => lib%entry(i))
                i_external = entry%i_external
                if (i_external /= 0 &
                     .and. lib%entry(i)%status == STAT_CONFIGURED) then
                   call lib%driver%clean_proc (i_external, os_data)
                end if
              end associate
              if (signal_is_pending ())  return
           end do
           call lib%driver%make_source (os_data)
        end if
        lib%status = STAT_SOURCE
        where (lib%entry%i_external /= 0 &
             .and. lib%entry%status == STAT_CONFIGURED)
           lib%entry%status = STAT_SOURCE
        end where
        lib%status = STAT_SOURCE
     end select
  end if
end subroutine process_library_make_source
```

Compile the generated code and update the status codes. Try to make the sources first, just in case. This includes compiling possible LaTeXFeynman diagram files.

⟨*Process libraries: process library: TBP*⟩+≡

1862

```
         procedure :: make_compile => process_library_make_compile
```

⟨*Process libraries: procedures*⟩+≡
```
      subroutine process_library_make_compile (lib, os_data, keep_old_source)
        class(process_library_t), intent(inout) :: lib
        type(os_data_t), intent(in) :: os_data
        logical, intent(in), optional :: keep_old_source
        if (lib%external .and. .not. lib%static) then
           select case (lib%status)
           case (STAT_CONFIGURED)
              call lib%make_source (os_data, keep_old_source)
           end select
           if (signal_is_pending ())  return
           select case (lib%status)
           case (STAT_SOURCE)
              call msg_message ("Process library '" // char (lib%basename) &
                   // "': compiling sources")
              call lib%driver%make_compile (os_data)
              where (lib%entry%i_external /= 0 &
                   .and. lib%entry%status == STAT_SOURCE)
                 lib%entry%status = STAT_COMPILED
              end where
              lib%status = STAT_COMPILED
           end select
        end if
      end subroutine process_library_make_compile
```

Link the process library. Try to compile first, just in case.

⟨*Process libraries: process library: TBP*⟩+≡
```
         procedure :: make_link => process_library_make_link
```

⟨*Process libraries: procedures*⟩+≡
```
      subroutine process_library_make_link (lib, os_data, keep_old_source)
        class(process_library_t), intent(inout) :: lib
        type(os_data_t), intent(in) :: os_data
        logical, intent(in), optional :: keep_old_source
        if (lib%external .and. .not. lib%static) then
           select case (lib%status)
           case (STAT_CONFIGURED:STAT_SOURCE)
              call lib%make_compile (os_data, keep_old_source)
           end select
           if (signal_is_pending ())  return
           select case (lib%status)
           case (STAT_COMPILED)
              call msg_message ("Process library '" // char (lib%basename) &
                   // "': linking")
              call lib%driver%make_link (os_data)
              lib%entry%status = STAT_LINKED
              lib%status = STAT_LINKED
           end select
        end if
      end subroutine process_library_make_link
```

Load the process library, i.e., assign pointers to the library functions.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: load => process_library_load
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_load (lib, os_data, keep_old_source)
      class(process_library_t), intent(inout) :: lib
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: keep_old_source
      select case (lib%status)
      case (STAT_CONFIGURED:STAT_COMPILED)
         call lib%make_link (os_data, keep_old_source)
      end select
      if (signal_is_pending ())  return
      select case (lib%status)
      case (STAT_LINKED)
         if (lib%external) then
            call msg_message ("Process library '" // char (lib%basename) &
                 // "': loading")
            call lib%driver%load (os_data)
            call lib%load_entries ()
         end if
         lib%entry%status = STAT_ACTIVE
         lib%status = STAT_ACTIVE
      end select
    end subroutine process_library_load
```

This is the actual loading part for the process methods.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: load_entries => process_library_load_entries
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_load_entries (lib)
      class(process_library_t), intent(inout) :: lib
      integer :: i
      do i = 1, size (lib%entry)
         associate (entry => lib%entry(i))
           if (entry%i_external /= 0) then
              call entry%connect (lib%driver, entry%i_external)
           end if
         end associate
      end do
    end subroutine process_library_load_entries
```

Unload the library, if possible. This reverts the status to "linked".

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: unload => process_library_unload
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_unload (lib)
      class(process_library_t), intent(inout) :: lib
      select case (lib%status)
      case (STAT_ACTIVE)
         if (lib%external) then
```

```
            call msg_message ("Process library '" // char (lib%basename) &
                  // "': unloading")
            call lib%driver%unload ()
         end if
         lib%entry%status = STAT_LINKED
         lib%status = STAT_LINKED
      end select
    end subroutine process_library_unload
```

Unload, clean all generated files and revert the library status. If `distclean` is set, also remove the makefile and the driver source.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: clean => process_library_clean
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_clean (lib, os_data, distclean)
      class(process_library_t), intent(inout) :: lib
      type(os_data_t), intent(in) :: os_data
      logical, intent(in) :: distclean
      call lib%unload ()
      if (lib%external .and. .not. lib%static) then
         call msg_message ("Process library '" // char (lib%basename) &
               // "': removing old files")
         if (distclean) then
            call lib%driver%distclean (os_data)
         else
            call lib%driver%clean (os_data)
         end if
      end if
      where (lib%entry%i_external /= 0)
         lib%entry%status = STAT_CONFIGURED
      elsewhere
         lib%entry%status = STAT_LINKED
      end where
      if (lib%external) then
         lib%status = STAT_CONFIGURED
      else
         lib%status = STAT_LINKED
      end if
    end subroutine process_library_clean
```

Unload and clean the compiled library and all generated object files, so force a recompilation.

⟨*XXX Process libraries: process library: TBP*⟩≡
```
    procedure :: clean_objects => process_library_clean_objects
```

⟨*XXX Process libraries: procedures*⟩≡
```
    subroutine process_library_clean_objects (lib, os_data)
      class(process_library_t), intent(inout) :: lib
      type(os_data_t), intent(in) :: os_data
      call lib%unload ()
      if (lib%external .and. .not. lib%static) then
         call msg_message ("Process library '" // char (lib%basename) &
               // "': removing compiled objects")
```

```
      call lib%driver%clean_library (os_data)
      call lib%driver%clean_objects (os_data)
   end if
   where (lib%entry%i_external /= 0)
      lib%entry%status = STAT_CONFIGURED
   elsewhere
      lib%entry%status = STAT_LINKED
   end where
   if (lib%external) then
      lib%status = STAT_CONFIGURED
   else
      lib%status = STAT_LINKED
   end if
end subroutine process_library_clean_objects
```

Unload and revert the library status to INITIAL. This allows for appending new processes. No files are deleted.

⟨*Process libraries: process library: TBP*⟩+≡
```
   procedure :: open => process_library_open
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_library_open (lib)
  class(process_library_t), intent(inout) :: lib
  select case (lib%status)
  case (STAT_OPEN)
  case default
     call lib%unload ()
     if (.not. lib%static) then
        lib%entry%status = STAT_OPEN
        lib%status = STAT_OPEN
        call msg_message ("Process library '" // char (lib%basename) &
             // "': open")
     else
        call msg_error ("Static process library '" // char (lib%basename) &
             // "': processes can't be appended")
     end if
  end select
end subroutine process_library_open
```

14.4.4 Use the library

Return the base name of the library

⟨*Process libraries: process library: TBP*⟩+≡
```
   procedure :: get_name => process_library_get_name
```

⟨*Process libraries: procedures*⟩+≡
```
function process_library_get_name (lib) result (name)
  class(process_library_t), intent(in) :: lib
  type(string_t) :: name
  name = lib%basename
end function process_library_get_name
```

Once activated, we view the process library object as an interface for accessing the matrix elements.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: is_active => process_library_is_active
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_is_active (lib) result (flag)
      logical :: flag
      class(process_library_t), intent(in) :: lib
      flag = lib%status == STAT_ACTIVE
    end function process_library_is_active
```

Retrieve constants using the process library driver. We assume that the process code has been loaded, if external.

⟨*Process libraries: process library entry: TBP*⟩+≡
```
    procedure :: fill_constants => process_library_entry_fill_constants
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_entry_fill_constants (entry, driver, data)
      class(process_library_entry_t), intent(in) :: entry
      class(prclib_driver_t), intent(in) :: driver
      type(process_constants_t), intent(out) :: data
      integer :: i
      if (entry%i_external /= 0) then
         i = entry%i_external
         data%id          = driver%get_process_id (i)
         data%model_name = driver%get_model_name (i)
         data%md5sum      = driver%get_md5sum (i)
         data%openmp_supported = driver%get_openmp_status (i)
         data%n_in  = driver%get_n_in  (i)
         data%n_out = driver%get_n_out (i)
         data%n_flv = driver%get_n_flv (i)
         data%n_hel = driver%get_n_hel (i)
         data%n_col = driver%get_n_col (i)
         data%n_cin = driver%get_n_cin (i)
         data%n_cf  = driver%get_n_cf  (i)
         call driver%set_flv_state (i, data%flv_state)
         call driver%set_hel_state (i, data%hel_state)
         call driver%set_col_state (i, data%col_state, data%ghost_flag)
         call driver%set_color_factors (i, data%color_factors, data%cf_index)
      else
         select type (proc_driver => entry%driver)
         class is (process_driver_internal_t)
            call proc_driver%fill_constants (data)
         end select
      end if
    end subroutine process_library_entry_fill_constants
```

Retrieve the constants and a connected driver for a process, identified by a process ID and a subprocess index. We scan the process entries until we have found a match.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: connect_process => process_library_connect_process
```

⟨*Process libraries: procedures*⟩+≡
```
    subroutine process_library_connect_process &
        (lib, id, i_component, data, proc_driver)
      class(process_library_t), intent(in) :: Lib
      type(string_t), intent(in) :: id
      integer, intent(in) :: i_component
      type(process_constants_t), intent(out) :: data
      class(prc_core_driver_t), allocatable, intent(out) :: proc_driver
      integer :: i
      do i = 1, size (lib%entry)
         associate (entry => lib%entry(i))
           if (entry%def%id == id .and. entry%i_component == i_component) then
              call entry%fill_constants (lib%driver, data)
              allocate (proc_driver, source=entry%driver)
              return
           end if
         end associate
      end do
      call msg_fatal ("Process library '" // char (lib%basename) &
                // "': process '" // char (id) // "' not found")
    end subroutine process_library_connect_process
```

14.4.5 Collect model-specific libraries

This returns appropriate linker flags for the model parameter libraries that are
used by the generated matrix element. At the end, the main libwhizard is
appended (again), because functions from that may be reqired.

Extra models in the local user space need to be treated individually.

⟨*Process libraries: process library: TBP*⟩+≡
```
    procedure :: get_modellibs_ldflags => process_library_get_modellibs_ldflags
```

⟨*Process libraries: procedures*⟩+≡
```
    function process_library_get_modellibs_ldflags (prc_lib, os_data) result (flags)
      class(process_library_t), intent(in) :: prc_lib
      type(os_data_t), intent(in) :: os_data
      type(string_t) :: flags
      type(string_t), dimension(:), allocatable :: models
      type(string_t) :: modelname, modellib, modellib_full
      logical :: exist
      integer :: i, j, mi
      flags = " -lomega"
      if ((.not. os_data%use_testfiles) .and. &
               os_dir_exist (os_data%whizard_models_libpath_local)) &
                 flags = flags // " -L" // os_data%whizard_models_libpath_local
      flags = flags // " -L" // os_data%whizard_models_libpath
      allocate (models(prc_lib%n_entries + 1))
      models = ""
      mi = 1
      if (allocated (prc_lib%entry)) then
         SCAN: do i = 1, prc_lib%n_entries
            if (associated (prc_lib%entry(i)%def)) then
               if (associated (prc_lib%entry(i)%def%model)) then
                  modelname = model_get_name (prc_lib%entry(i)%def%model)
```

1868

```
                else
                    cycle SCAN
                end if
            else
                cycle SCAN
            end if
            do j = 1, mi
                if (models(mi) == modelname) cycle SCAN
            end do
            models(mi) = modelname
            mi = mi + 1
            if (os_data%use_libtool) then
                modellib = "libparameters_" // modelname // ".la"
            else
                modellib = "libparameters_" // modelname // ".a"
            end if
            exist = .false.
            if (.not. os_data%use_testfiles) then
                modellib_full = os_data%whizard_models_libpath_local &
                        // "/" // modellib
                inquire (file=char (modellib_full), exist=exist)
            end if
            if (.not. exist) then
                modellib_full = os_data%whizard_models_libpath &
                        // "/" // modellib
                inquire (file=char (modellib_full), exist=exist)
            end if
            if (exist) flags = flags // " -lparameters_" // modelname
        end do SCAN
    end if
    deallocate (models)
    flags = flags // " -lwhizard"
end function process_library_get_modellibs_ldflags
```

14.4.6 Test

This is the master for calling self-test procedures.

⟨*Process libraries: public*⟩+≡
```
    public :: process_libraries_test
```

⟨*Process libraries: tests*⟩≡
```
    subroutine process_libraries_test (u, results)
        integer, intent(in) :: u
        type(test_results_t), intent(inout) :: results
```
⟨*Process libraries: execute tests*⟩
```
    end subroutine process_libraries_test
```

Empty process list

Test 1: Write an empty process list.

⟨*Process libraries: execute tests*⟩≡

```
    call test (process_libraries_1, "process_libraries_1", &
        "empty process list", &
        u, results)
```

⟨Process libraries: tests⟩+≡
```
    subroutine process_libraries_1 (u)
      integer, intent(in) :: u
      type(process_def_list_t) :: process_def_list

      write (u, "(A)")  "* Test output: process_libraries_1"
      write (u, "(A)")  "*   Purpose: Display an empty process definition list"
      write (u, "(A)")

      call process_def_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_libraries_1"
    end subroutine process_libraries_1
```

Process definition list

Test 2: Process definition list with processes and components. Construct the list, write to file, read it in again, and display. Finalize and delete the list after use.

We define a trivial 'test' type for the process variant. The test type contains just one (meaningless) data item, which is an integer.

⟨Process libraries: test declarations⟩≡
```
    type, extends (prc_core_def_t) :: prcdef_2_t
      integer :: data = 0
      logical :: file = .false.
     contains
     ⟨Process libraries: prcdef 2: TBP⟩
    end type prcdef_2_t
```

The process variant is named 'test'.

⟨Process libraries: prcdef 2: TBP⟩≡
```
    procedure, nopass :: type_string => prcdef_2_type_string
```

⟨Process libraries: tests⟩+≡
```
    function prcdef_2_type_string () result (string)
      type(string_t) :: string
      string = "test"
    end function prcdef_2_type_string
```

Write the contents (the integer value).

⟨Process libraries: prcdef 2: TBP⟩+≡
```
    procedure :: write => prcdef_2_write
```

⟨Process libraries: tests⟩+≡
```
    subroutine prcdef_2_write (object, unit)
      class(prcdef_2_t), intent(in) :: object
      integer, intent(in) :: unit
```

```
      write (unit, "(3x,A,I0)")  "Test data          = ", object%data
    end subroutine prcdef_2_write
```

Recover the integer value.

⟨*Process libraries: prcdef 2: TBP*⟩+≡
```
    procedure :: read => prcdef_2_read
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_2_read (object, unit)
      class(prcdef_2_t), intent(out) :: object
      integer, intent(in) :: unit
      character(80) :: buffer
      read (unit, "(A)")  buffer
      call strip_prefix (buffer)
      read (buffer, *)  object%data
    end subroutine prcdef_2_read
```

No external procedures.

⟨*Process libraries: prcdef 2: TBP*⟩+≡
```
    procedure, nopass :: get_features => prcdef_2_get_features
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_2_get_features (features)
      type(string_t), dimension(:), allocatable, intent(out) :: features
      allocate (features (0))
    end subroutine prcdef_2_get_features
```

No code generated.

⟨*Process libraries: prcdef 2: TBP*⟩+≡
```
    procedure :: generate_code => prcdef_2_generate_code
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_2_generate_code (object, &
         basename, model_name, prt_in, prt_out)
      class(prcdef_2_t), intent(in) :: object
      type(string_t), intent(in) :: basename
      type(string_t), intent(in) :: model_name
      type(string_t), dimension(:), intent(in) :: prt_in
      type(string_t), dimension(:), intent(in) :: prt_out
    end subroutine prcdef_2_generate_code
```

Allocate the driver with the appropriate type.

⟨*Process libraries: prcdef 2: TBP*⟩+≡
```
    procedure :: allocate_driver => prcdef_2_allocate_driver
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_2_allocate_driver (object, driver, basename)
      class(prcdef_2_t), intent(in) :: object
      class(prc_core_driver_t), intent(out), allocatable :: driver
      type(string_t), intent(in) :: basename
      allocate (prctest_2_t :: driver)
    end subroutine prcdef_2_allocate_driver
```

Nothing to connect.

⟨Process libraries: prcdef 2: TBP⟩+≡
```
procedure :: connect => prcdef_2_connect
```

⟨Process libraries: tests⟩+≡
```
subroutine prcdef_2_connect (def, lib_driver, i, proc_driver)
  class(prcdef_2_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
end subroutine prcdef_2_connect
```

The associated driver type.

⟨Process libraries: types⟩+≡
```
type, extends (process_driver_internal_t) :: prctest_2_t
 contains
 ⟨Process libraries: prctest 2: TBP⟩
end type prctest_2_t
```

Return the type name.

⟨Process libraries: prctest 2: TBP⟩≡
```
procedure, nopass :: type_name => prctest_2_type_name
```

⟨Process libraries: procedures⟩+≡
```
function prctest_2_type_name () result (type)
  type(string_t) :: type
  type = "test"
end function prctest_2_type_name
```

This should fill constant process data. We do not check those here, however, therefore nothing done.

⟨Process libraries: prctest 2: TBP⟩+≡
```
procedure :: fill_constants => prctest_2_fill_constants
```

⟨Process libraries: procedures⟩+≡
```
subroutine prctest_2_fill_constants (driver, data)
  class(prctest_2_t), intent(in) :: driver
  type(process_constants_t), intent(out) :: data
end subroutine prctest_2_fill_constants
```

Here is the actual test.

For reading, we need a list of templates, i.e., an array containing allocated objects for all available process variants. This is the purpose of `process_core_templates`. Here, we have only a single template for the 'test' variant.

⟨Process libraries: execute tests⟩+≡
```
call test (process_libraries_2, "process_libraries_2", &
     "process definition list", &
     u, results)
```

⟨*Process libraries: tests*⟩+≡

```
    subroutine process_libraries_2 (u)
      integer, intent(in) :: u
      type(prc_template_t), dimension(:), allocatable :: process_core_templates
      type(process_def_list_t) :: process_def_list
      type(process_def_entry_t), pointer :: entry => null ()
      class(prc_core_def_t), allocatable :: test_def
      integer :: scratch_unit

      write (u, "(A)")  "* Test output: process_libraries_2"
      write (u, "(A)")  "* Purpose: Construct a process definition list,"
      write (u, "(A)")  "*          write it to file and reread it"
      write (u, "(A)")  ""
      write (u, "(A)")  "* Construct a process definition list"
      write (u, "(A)")  "*   First process definition: empty"
      write (u, "(A)")  "*   Second process definition: two components"
      write (u, "(A)")  "*     First component: empty"
      write (u, "(A)")  "*     Second component: test data"
      write (u, "(A)")  "*   Third process definition:"
      write (u, "(A)")  "*     Embedded decays and polarization"
      write (u, "(A)")

      allocate (process_core_templates (1))
      allocate (prcdef_2_t :: process_core_templates(1)%core_def)

      allocate (entry)
      call entry%init (var_str ("first"), n_in = 0, n_components = 0)
      call entry%compute_md5sum ()
      call process_def_list%append (entry)

      allocate (entry)
      call entry%init (var_str ("second"), model_name = var_str ("Test"), &
           n_in = 1, n_components = 2)
      allocate (prcdef_2_t :: test_def)
      select type (test_def)
      type is (prcdef_2_t);  test_def%data = 42
      end select
      call entry%import_component (2, n_out = 2, &
           prt_in  = new_prt_spec ([var_str ("a")]), &
           prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
           method  = var_str ("test"), &
           variant = test_def)
      call entry%compute_md5sum ()
      call process_def_list%append (entry)

      allocate (entry)
      call entry%init (var_str ("third"), model_name = var_str ("Test"), &
           n_in = 2, n_components = 1)
      allocate (prcdef_2_t :: test_def)
      call entry%import_component (1, n_out = 3, &
           prt_in  = &
             new_prt_spec ([var_str ("a"), var_str ("b")]), &
           prt_out = &
             [new_prt_spec (var_str ("c")), &
```

```
                new_prt_spec (var_str ("d"), .true.), &
                new_prt_spec (var_str ("e"), [var_str ("e_decay")])]], &
            method  = var_str ("test"), &
            variant = test_def)
    call entry%compute_md5sum ()
    call process_def_list%append (entry)
    call process_def_list%write (u)

    write (u, "(A)")  ""
    write (u, "(A)")  "* Write the process definition list to (scratch) file"

    scratch_unit = free_unit ()
    open (unit = scratch_unit, status="scratch", action = "readwrite")
    call process_def_list%write (scratch_unit)
    call process_def_list%final ()

    write (u, "(A)")  "* Reread it"
    write (u, "(A)")  ""

    rewind (scratch_unit)
    call process_def_list%read (scratch_unit, process_core_templates)
    close (scratch_unit)

    call process_def_list%write (u)
    call process_def_list%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_libraries_2"
  end subroutine process_libraries_2
```

Process library object

Test 3: Process library object with several process definitions and library entries. Just construct the object, modify some initial content, and write the result. The modifications are mostly applied directly, so we do not test anything but the contents and the output routine.

⟨*Process libraries: execute tests*⟩+≡
```
    call test (process_libraries_3, "process_libraries_3", &
        "recover process definition list from file", &
        u, results)
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine process_libraries_3 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      type(process_def_entry_t), pointer :: entry

      write (u, "(A)")  "* Test output: process_libraries_3"
      write (u, "(A)")  "* Purpose: Construct a process library object &
          &with entries"
      write (u, "(A)")  ""
      write (u, "(A)")  "* Construct and display a process library object"
      write (u, "(A)")  "*   with 5 entries"
```

```
      write (u, "(A)")  "*   associated with 3 matrix element codes"
      write (u, "(A)")  "*   corresponding to 3 process definitions"
      write (u, "(A)")  "*   with 2, 1, 1 components, respectively"
      write (u, "(A)")

      call lib%init (var_str ("testlib"))

      lib%status = STAT_ACTIVE
      lib%n_entries = 5
      allocate (lib%entry (lib%n_entries))

      allocate (entry)
      call entry%init (var_str ("test_a"), n_in = 2, n_components = 2)
      call lib%entry(3)%init (STAT_SOURCE, entry%process_def_t, 2, 2)
      allocate (prctest_2_t :: lib%entry(3)%driver)
      call lib%entry(4)%init (STAT_COMPILED, entry%process_def_t, 1, 0)
      call lib%append (entry)

      allocate (entry)
      call entry%init (var_str ("test_b"), n_in = 2, n_components = 1)
      call lib%entry(2)%init (STAT_CONFIGURED, entry%process_def_t, 1, 1)
      call lib%append (entry)

      allocate (entry)
      call entry%init (var_str ("test_c"), n_in = 2, n_components = 1)
      call lib%entry(5)%init (STAT_LINKED, entry%process_def_t, 1, 3)
      allocate (prctest_2_t :: lib%entry(5)%driver)
      call lib%append (entry)

      call lib%write (u)
      call lib%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_libraries_3"
    end subroutine process_libraries_3
```

Process library for test matrix element (no file)

Test 4: We proceed through the library generation and loading phases with a test matrix element type that needs no code written on file.

⟨*Process libraries: execute tests*⟩+≡
```
    call test (process_libraries_4, "process_libraries_4", &
        "build and load internal process library", &
        u, results)
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine process_libraries_4 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      type(process_def_entry_t), pointer :: entry
      class(prc_core_def_t), allocatable :: core_def
      type(os_data_t) :: os_data
```

```
write (u, "(A)")  "* Test output: process_libraries_4"
write (u, "(A)")  "* Purpose: build a process library with an &
     &internal (pseudo) matrix element"
write (u, "(A)")  "*          No Makefile or code should be generated"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library with one entry &
     &(no external code)"
write (u, "(A)")
call os_data_init (os_data)
call lib%init (var_str ("proclibs4"))

allocate (prcdef_2_t :: core_def)

allocate (entry)
call entry%init (var_str ("proclibs4_a"), n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Compute MD5 sum"
write (u, "(A)")
call lib%compute_md5sum ()

write (u, "(A)")  "* Write makefile (no-op)"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Write driver source code (no-op)"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code (no-op)"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(A)")  "* Compile (no-op)"
write (u, "(A)")
call lib%make_compile (os_data)

write (u, "(A)")  "* Link (no-op)"
write (u, "(A)")
call lib%make_link (os_data)

write (u, "(A)")  "* Load (no-op)"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)
call lib%final ()
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Test output end: process_libraries_4"
   end subroutine process_libraries_4
```

Build workflow for test matrix element

Test 5: We write source code for a dummy process.

We define another trivial type for the process variant. The test type contains just no variable data, but produces code on file.

⟨*Process libraries: test declarations*⟩+≡
```
   type, extends (prc_core_def_t) :: prcdef_5_t
    contains
    ⟨Process libraries: prcdef 5: TBP⟩
   end type prcdef_5_t
```

The process variant is named `test_file`.

⟨*Process libraries: prcdef 5: TBP*⟩≡
```
   procedure, nopass :: type_string => prcdef_5_type_string
```

⟨*Process libraries: tests*⟩+≡
```
   function prcdef_5_type_string () result (string)
     type(string_t) :: string
     string = "test_file"
   end function prcdef_5_type_string
```

We reuse the writer `test_writer_4` from the previous module.

⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
   procedure :: init => prcdef_5_init
```

⟨*Process libraries: tests*⟩+≡
```
   subroutine prcdef_5_init (object)
     class(prcdef_5_t), intent(out) :: object
     allocate (test_writer_4_t :: object%writer)
   end subroutine prcdef_5_init
```

Nothing to write.

⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
   procedure :: write => prcdef_5_write
```

⟨*Process libraries: tests*⟩+≡
```
   subroutine prcdef_5_write (object, unit)
     class(prcdef_5_t), intent(in) :: object
     integer, intent(in) :: unit
   end subroutine prcdef_5_write
```

Nothing to read.

⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
   procedure :: read => prcdef_5_read
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_5_read (object, unit)
      class(prcdef_5_t), intent(out) :: object
      integer, intent(in) :: unit
    end subroutine prcdef_5_read
```

Allocate the driver with the appropriate type.
⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
    procedure :: allocate_driver => prcdef_5_allocate_driver
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_5_allocate_driver (object, driver, basename)
      class(prcdef_5_t), intent(in) :: object
      class(prc_core_driver_t), intent(out), allocatable :: driver
      type(string_t), intent(in) :: basename
      allocate (prctest_5_t :: driver)
    end subroutine prcdef_5_allocate_driver
```

This time we need code:
⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
    procedure, nopass :: needs_code => prcdef_5_needs_code
```

⟨*Process libraries: tests*⟩+≡
```
    function prcdef_5_needs_code () result (flag)
      logical :: flag
      flag = .true.
    end function prcdef_5_needs_code
```

For the test case, we implement a single feature `proc1`.
⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
    procedure, nopass :: get_features => prcdef_5_get_features
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_5_get_features (features)
      type(string_t), dimension(:), allocatable, intent(out) :: features
      allocate (features (1))
      features = [ var_str ("proc1") ]
    end subroutine prcdef_5_get_features
```

Nothing to connect.
⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
    procedure :: connect => prcdef_5_connect
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine prcdef_5_connect (def, lib_driver, i, proc_driver)
      class(prcdef_5_t), intent(in) :: def
      class(prclib_driver_t), intent(in) :: lib_driver
      integer, intent(in) :: i
      class(prc_core_driver_t), intent(inout) :: proc_driver
    end subroutine prcdef_5_connect
```

The driver type.

⟨*Process libraries: types*⟩+≡
```
type, extends (prc_core_driver_t) :: prctest_5_t
 contains
 ⟨Process libraries: prctest 5: TBP⟩
end type prctest_5_t
```

Return the type name.

⟨*Process libraries: prctest 5: TBP*⟩≡
```
procedure, nopass :: type_name => prctest_5_type_name
```

⟨*Process libraries: tests*⟩+≡
```
function prctest_5_type_name () result (type)
  type(string_t) :: type
  type = "test_file"
end function prctest_5_type_name
```

Here is the actual test:

⟨*Process libraries: execute tests*⟩+≡
```
call test (process_libraries_5, "process_libraries_5", &
     "build external process library", &
     u, results)
```

⟨*Process libraries: tests*⟩+≡
```
subroutine process_libraries_5 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
  class(prc_core_def_t), allocatable :: core_def
  type(os_data_t) :: os_data

  write (u, "(A)")  "* Test output: process_libraries_5"
  write (u, "(A)")  "* Purpose: build a process library with an &
       &external (pseudo) matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("proclibs5"))
  call os_data_init (os_data)

  allocate (prcdef_5_t :: core_def)
  select type (core_def)
  type is (prcdef_5_t)
     call core_def%init ()
  end select

  allocate (entry)
  call entry%init (var_str ("proclibs5_a"), &
       model_name = var_str ("Test_Model"), &
       n_in = 1, n_components = 1)
  call entry%import_component (1, n_out = 2, &
       prt_in  = new_prt_spec ([var_str ("a")]), &
       prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
```

1879

```
      method  = var_str ("test"), &
      variant = core_def)
  call lib%append (entry)

  write (u, "(A)")  "* Configure library"
  write (u, "(A)")
  call lib%configure (os_data)

  write (u, "(A)")  "* Compute MD5 sum"
  write (u, "(A)")
  call lib%compute_md5sum ()

  write (u, "(A)")  "* Write makefile"
  write (u, "(A)")
  call lib%write_makefile (os_data, force = .true.)

  write (u, "(A)")  "* Write driver source code"
  write (u, "(A)")
  call lib%write_driver (force = .true.)

  write (u, "(A)")  "* Write process source code"
  write (u, "(A)")
  call lib%make_source (os_data)

  write (u, "(A)")  "* Compile"
  write (u, "(A)")
  call lib%make_compile (os_data)

  write (u, "(A)")  "* Link"
  write (u, "(A)")
  call lib%make_link (os_data)

  call lib%write (u, libpath = .false.)

  call lib%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_libraries_5"
end subroutine process_libraries_5
```

Build and load library with test matrix element

Test 6: We write source code for a dummy process.

This process variant is identical to the previous case, but it supports a driver for the test procedure 'proc1'.

⟨*Process libraries: test declarations*⟩+≡
```
    type, extends (prc_core_def_t) :: prcdef_6_t
     contains
     ⟨Process libraries: prcdef 6: TBP⟩
    end type prcdef_6_t
```

The process variant is named `test_file`.

⟨Process libraries: prcdef 6: TBP⟩≡
```
procedure, nopass :: type_string => prcdef_6_type_string
```

⟨Process libraries: tests⟩+≡
```
function prcdef_6_type_string () result (string)
  type(string_t) :: string
  string = "test_file"
end function prcdef_6_type_string
```

We reuse the writer `test_writer_4` from the previous module.

⟨Process libraries: prcdef 6: TBP⟩+≡
```
procedure :: init => prcdef_6_init
```

⟨Process libraries: tests⟩+≡
```
subroutine prcdef_6_init (object)
  class(prcdef_6_t), intent(out) :: object
  allocate (test_writer_4_t :: object%writer)
  call object%writer%init_test ()
end subroutine prcdef_6_init
```

Nothing to write.

⟨Process libraries: prcdef 6: TBP⟩+≡
```
procedure :: write => prcdef_6_write
```

⟨Process libraries: tests⟩+≡
```
subroutine prcdef_6_write (object, unit)
  class(prcdef_6_t), intent(in) :: object
  integer, intent(in) :: unit
end subroutine prcdef_6_write
```

Nothing to read.

⟨Process libraries: prcdef 6: TBP⟩+≡
```
procedure :: read => prcdef_6_read
```

⟨Process libraries: tests⟩+≡
```
subroutine prcdef_6_read (object, unit)
  class(prcdef_6_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine prcdef_6_read
```

Allocate the driver with the appropriate type.

⟨Process libraries: prcdef 6: TBP⟩+≡
```
procedure :: allocate_driver => prcdef_6_allocate_driver
```

⟨Process libraries: tests⟩+≡
```
subroutine prcdef_6_allocate_driver (object, driver, basename)
  class(prcdef_6_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (prctest_6_t :: driver)
end subroutine prcdef_6_allocate_driver
```

This time we need code:

⟨*Process libraries: prcdef 6: TBP*⟩+≡
```
procedure, nopass :: needs_code => prcdef_6_needs_code
```

⟨*Process libraries: tests*⟩+≡
```
function prcdef_6_needs_code () result (flag)
  logical :: flag
  flag = .true.
end function prcdef_6_needs_code
```

For the test case, we implement a single feature `proc1`.

⟨*Process libraries: prcdef 6: TBP*⟩+≡
```
procedure, nopass :: get_features => prcdef_6_get_features
```

⟨*Process libraries: tests*⟩+≡
```
subroutine prcdef_6_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (1))
  features = [ var_str ("proc1") ]
end subroutine prcdef_6_get_features
```

The interface of the only specific feature.

⟨*Process libraries: interfaces*⟩≡
```
abstract interface
  subroutine proc1_t (n) bind(C)
    import
    integer(c_int), intent(out) :: n
  end subroutine proc1_t
end interface
```

Connect the feature `proc1` with the process driver.

⟨*Process libraries: prcdef 6: TBP*⟩+≡
```
procedure :: connect => prcdef_6_connect
```

⟨*Process libraries: tests*⟩+≡
```
subroutine prcdef_6_connect (def, lib_driver, i, proc_driver)
  class(prcdef_6_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  integer(c_int) :: pid, fid
  type(c_funptr) :: fptr
  select type (proc_driver)
  type is  (prctest_6_t)
    pid = i
    fid = 1
    call lib_driver%get_fptr (pid, fid, fptr)
    call c_f_procpointer (fptr, proc_driver%proc1)
  end select
end subroutine prcdef_6_connect
```

The driver type.

⟨*Process libraries: types*⟩+≡
```
type, extends (prc_core_driver_t) :: prctest_6_t
  procedure(proc1_t), nopass, pointer :: proc1 => null ()
 contains
 ⟨Process libraries: prctest 6: TBP⟩
 end type prctest_6_t
```

Return the type name.

⟨*Process libraries: prctest 6: TBP*⟩≡
```
procedure, nopass :: type_name => prctest_6_type_name
```

⟨*Process libraries: tests*⟩+≡
```
function prctest_6_type_name () result (type)
  type(string_t) :: type
  type = "test_file"
end function prctest_6_type_name
```

Here is the actual test:

⟨*Process libraries: execute tests*⟩+≡
```
call test (process_libraries_6, "process_libraries_6", &
     "build and load external process library", &
     u, results)
```

⟨*Process libraries: tests*⟩+≡
```
subroutine process_libraries_6 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
  class(prc_core_def_t), allocatable :: core_def
  type(os_data_t) :: os_data
  type(string_t), dimension(:), allocatable :: name_list
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: proc_driver
  integer :: i
  integer(c_int) :: n

  write (u, "(A)")  "* Test output: process_libraries_6"
  write (u, "(A)")  "* Purpose: build and load a process library"
  write (u, "(A)")  "*          with an external (pseudo) matrix element"
  write (u, "(A)")  "*          Check single-call linking"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("proclibs6"))
  call os_data_init (os_data)

  allocate (prcdef_6_t :: core_def)
  select type (core_def)
  type is (prcdef_6_t)
     call core_def%init ()
  end select
```

1883

```
allocate (entry)
call entry%init (var_str ("proclibs6_a"), &
     model_name = var_str ("Test_model"), &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
     method  = var_str ("test"), &
     variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,A,A)")  "name                        = '", &
     char (lib%get_name ()), "'"
write (u, "(1x,A,L1)")  "is active                = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
     lib%get_n_processes ()
write (u, "(1x,A)", advance="no")  "processes               ="
call lib%get_process_id_list (name_list)
do i = 1, size (name_list)
   write (u, "(1x,A)", advance="no")  char (name_list(i))
end do
write (u, *)
write (u, "(1x,A,L1)")  "proclibs6_a is process   = ", &
     lib%contains (var_str ("proclibs6_a"))
write (u, "(1x,A,I0)")  "proclibs6_a has index    = ", &
     lib%get_entry_index (var_str ("proclibs6_a"))
write (u, "(1x,A,L1)")  "foobar is process        = ", &
     lib%contains (var_str ("foobar"))
write (u, "(1x,A,I0)")  "foobar has index         = ", &
     lib%get_entry_index (var_str ("foobar"))
write (u, "(1x,A,I0)")  "n_in(proclibs6_a)        = ", &
```

```
       lib%get_n_in (var_str ("proclibs6_a"))
  write (u, "(1x,A,A)")   "model_name(proclibs6_a)   = ", &
       char (lib%get_model_name (var_str ("proclibs6_a")))
  write (u, "(1x,A,I0)")  "n_components(proclibs6_a) = ", &
       lib%get_n_components (var_str ("proclibs6_a"))
  write (u, "(1x,A)", advance="no")  "components(proclibs6_a)   ="
  call lib%get_component_list (var_str ("proclibs6_a"), name_list)
  do i = 1, size (name_list)
     write (u, "(1x,A)", advance="no")  char (name_list(i))
  end do
  write (u, *)

  write (u, "(A)")
  write (u, "(A)")   "* Constants of proclibs6_a_i1:"
  write (u, "(A)")

  call lib%connect_process (var_str ("proclibs6_a"), 1, data, proc_driver)

  write (u, "(1x,A,A)")   "component ID     = ", char (data%id)
  write (u, "(1x,A,A)")   "model name       = ", char (data%model_name)
  write (u, "(1x,A,A,A)") "md5sum           = '", data%md5sum, "'"
  write (u, "(1x,A,L1)")  "openmp supported = ", data%openmp_supported
  write (u, "(1x,A,I0)")  "n_in  = ", data%n_in
  write (u, "(1x,A,I0)")  "n_out = ", data%n_out
  write (u, "(1x,A,I0)")  "n_flv = ", data%n_flv
  write (u, "(1x,A,I0)")  "n_hel = ", data%n_hel
  write (u, "(1x,A,I0)")  "n_col = ", data%n_col
  write (u, "(1x,A,I0)")  "n_cin = ", data%n_cin
  write (u, "(1x,A,I0)")  "n_cf  = ", data%n_cf
  write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
  write (u, "(1x,A,10(1x,I0))") "hel state =", data%hel_state
  write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
  write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
  write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
  write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

  write (u, "(A)")
  write (u, "(A)")   "* Call feature of proclibs6_a:"
  write (u, "(A)")

  select type (proc_driver)
  type is (prctest_6_t)
     call proc_driver%proc1 (n)
     write (u, "(1x,A,I0)") "proc1 = ", n
  end select

  call lib%final ()

  write (u, "(A)")
  write (u, "(A)")   "* Test output end: process_libraries_6"
end subroutine process_libraries_6
```

MD5 sums

Check MD5 sum calculation.

⟨*Process libraries: execute tests*⟩+≡
```
    call test (process_libraries_7, "process_libraries_7", &
        "process definition list", &
        u, results)
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine process_libraries_7 (u)
      integer, intent(in) :: u
      type(prc_template_t), dimension(:), allocatable :: process_core_templates
      type(process_def_entry_t) :: entry
      class(prc_core_def_t), allocatable :: test_def

      write (u, "(A)")  "* Test output: process_libraries_7"
      write (u, "(A)")  "* Purpose: Construct a process definition list &
          &and check MD5 sums"
      write (u, "(A)")
      write (u, "(A)")  "* Construct a process definition list"
      write (u, "(A)")  "*   Process: two components"
      write (u, "(A)")

      allocate (process_core_templates (1))
      allocate (prcdef_2_t :: process_core_templates(1)%core_def)

      call entry%init (var_str ("first"), model_name = var_str ("Test"), &
          n_in = 1, n_components = 2)
      allocate (prcdef_2_t :: test_def)
      select type (test_def)
      type is (prcdef_2_t);  test_def%data = 31
      end select
      call entry%import_component (1, n_out = 3, &
          prt_in  = new_prt_spec ([var_str ("a")]), &
          prt_out = new_prt_spec ([var_str ("b"), var_str ("c"), &
                                   var_str ("e")]), &
          method  = var_str ("test"), &
          variant = test_def)
      allocate (prcdef_2_t :: test_def)
      select type (test_def)
      type is (prcdef_2_t);  test_def%data = 42
      end select
      call entry%import_component (2, n_out = 2, &
          prt_in  = new_prt_spec ([var_str ("a")]), &
          prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
          method  = var_str ("test"), &
          variant = test_def)
      call entry%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Compute MD5 sums"
      write (u, "(A)")

      call entry%compute_md5sum ()
      call entry%write (u)
```

```
      write (u, "(A)")
      write (u, "(A)")  "* Recalculate MD5 sums (should be identical)"
      write (u, "(A)")

      call entry%compute_md5sum ()
      call entry%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Modify a component and recalculate MD5 sums"
      write (u, "(A)")

      select type (test_def => entry%initial(2)%core_def)
      type is (prcdef_2_t)
         test_def%data = 54
      end select
      call entry%compute_md5sum ()
      call entry%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Modify the model and recalculate MD5 sums"
      write (u, "(A)")

      entry%model_name = "foo"
      call entry%compute_md5sum ()
      call entry%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_libraries_7"
    end subroutine process_libraries_7
```

Here is the actual test:

⟨*Process libraries: execute tests*⟩+≡
```
    call test (process_libraries_8, "process_libraries_8", &
        "library status checks", &
        u, results)
```

⟨*Process libraries: tests*⟩+≡
```
    subroutine process_libraries_8 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      type(process_def_entry_t), pointer :: entry
      class(prc_core_def_t), allocatable :: core_def
      type(os_data_t) :: os_data

      write (u, "(A)")  "* Test output: process_libraries_8"
      write (u, "(A)")  "* Purpose: build and load a process library"
      write (u, "(A)")  "*          with an external (pseudo) matrix element"
      write (u, "(A)")  "*          Check status updates"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize a process library with one entry"
      write (u, "(A)")
      call lib%init (var_str ("proclibs8"))
```

1887

```
call os_data_init (os_data)

allocate (prcdef_6_t :: core_def)
select type (core_def)
type is (prcdef_6_t)
   call core_def%init ()
end select

allocate (entry)
call entry%init (var_str ("proclibs8_a"), &
     model_name = var_str ("Test_model"), &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
     method  = var_str ("test"), &
     variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")

call lib%configure (os_data)
call lib%compute_md5sum ()

associate (def => lib%entry(1)%def%initial(1))
  select type (writer => lib%driver%record(1)%writer)
  type is (test_writer_4_t)
     writer%md5sum = def%md5sum
  end select
end associate

write (u, "(1x,A,L1)")  "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")  "lib status   = ", lib%status
write (u, "(1x,A,I0)")  "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)")  "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")  "lib status   = ", lib%status
write (u, "(1x,A,I0)")  "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .false.)
```

```
write (u, "(A)")  "* Write process source code"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(1x,A,L1)")  "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")  "lib status   = ", lib%status
write (u, "(1x,A,I0)")  "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Compile and load"
write (u, "(A)")

call lib%load (os_data)
write (u, "(1x,A,L1)")  "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")  "lib status   = ", lib%status
write (u, "(1x,A,I0)")  "proc1 status = ", lib%entry(1)%status

write (u, "(A)")
write (u, "(A)")  "* Append process and reconfigure"
write (u, "(A)")

allocate (prcdef_6_t :: core_def)
select type (core_def)
type is (prcdef_6_t)
   call core_def%init ()
end select

allocate (entry)
call entry%init (var_str ("proclibs8_b"), &
     model_name = var_str ("Test_model"), &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("d")]), &
     method  = var_str ("test"), &
     variant = core_def)
call lib%append (entry)

call lib%configure (os_data)
call lib%compute_md5sum ()
associate (def => lib%entry(2)%def%initial(1))
  select type (writer => lib%driver%record(2)%writer)
  type is (test_writer_4_t)
     writer%md5sum = def%md5sum
  end select
end associate
call lib%write_makefile (os_data, force = .false.)
call lib%write_driver (force = .false.)

write (u, "(1x,A,L1)")  "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")  "lib status   = ", lib%status
write (u, "(1x,A,I0)")  "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)")  "proc2 status = ", lib%entry(2)%status
```

```
write (u, "(A)")
write (u, "(A)")   "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)")   "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")   "lib status   = ", lib%status
write (u, "(1x,A,I0)")   "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)")   "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")   "* Write source code"
write (u, "(A)")

call lib%make_source (os_data)
write (u, "(1x,A,L1)")   "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")   "lib status   = ", lib%status
write (u, "(1x,A,I0)")   "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)")   "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")   "* Reset status"
write (u, "(A)")

lib%status = STAT_CONFIGURED
lib%entry%status = STAT_CONFIGURED
write (u, "(1x,A,L1)")   "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")   "lib status   = ", lib%status
write (u, "(1x,A,I0)")   "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)")   "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")   "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)")   "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")   "lib status   = ", lib%status
write (u, "(1x,A,I0)")   "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)")   "proc2 status = ", lib%entry(2)%status

write (u, "(A)")
write (u, "(A)")   "* Partial cleanup"
write (u, "(A)")

call lib%clean (os_data, distclean = .false.)
write (u, "(1x,A,L1)")   "library loaded = ", lib%driver%loaded
write (u, "(1x,A,I0)")   "lib status   = ", lib%status
write (u, "(1x,A,I0)")   "proc1 status = ", lib%entry(1)%status
write (u, "(1x,A,I0)")   "proc2 status = ", lib%entry(2)%status


write (u, "(A)")
write (u, "(A)")   "* Update status"
```

```
      write (u, "(A)")

      call lib%update_status (os_data)
      write (u, "(1x,A,L1)")  "library loaded = ", lib%driver%loaded
      write (u, "(1x,A,I0)")  "lib status   = ", lib%status
      write (u, "(1x,A,I0)")  "proc1 status = ", lib%entry(1)%status
      write (u, "(1x,A,I0)")  "proc2 status = ", lib%entry(2)%status

      write (u, "(A)")
      write (u, "(A)")  "* Complete cleanup"

      call lib%clean (os_data, distclean = .true.)
      call lib%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_libraries_8"
    end subroutine process_libraries_8
```

14.5   Process Library Stacks

For storing and handling multiple libraries, we define process library stacks.
These are ordinary stacks where new entries are pushed onto the top.

⟨prclib_stacks.f90⟩≡
  ⟨*File header*⟩

```
  module prclib_stacks
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use unit_tests
    use os_interface
    use models

    use process_libraries
```

  ⟨*Standard module head*⟩

  ⟨*Prclib stacks: public*⟩

  ⟨*Prclib stacks: types*⟩

```
  contains
```

  ⟨*Prclib stacks: procedures*⟩

  ⟨*Prclib stacks: tests*⟩

```
  end module prclib_stacks
```

14.5.1 The stack entry type

A stack entry is a process library object, augmented by a pointer to the next entry. We do not need specific methods, all relevant methods are inherited.

On higher level, process libraries should be prepared as process entry objects.

⟨*Prclib stacks: public*⟩≡
```
    public :: prclib_entry_t
```

⟨*Prclib stacks: types*⟩≡
```
    type, extends (process_library_t) :: prclib_entry_t
       type(prclib_entry_t), pointer :: next => null ()
    end type prclib_entry_t
```

14.5.2 The prclib stack type

For easy conversion and lookup it is useful to store the filling number in the object. The content is stored as a linked list.

⟨*Prclib stacks: public*⟩+≡
```
    public :: prclib_stack_t
```

⟨*Prclib stacks: types*⟩+≡
```
    type :: prclib_stack_t
       integer :: n = 0
       type(prclib_entry_t), pointer :: first => null ()
     contains
     ⟨Prclib stacks: prclib stack: TBP⟩
    end type prclib_stack_t
```

Finalizer. Iteratively deallocate the stack entries. The resulting empty stack can be immediately recycled, if necessary.

⟨*Prclib stacks: prclib stack: TBP*⟩≡
```
    procedure :: final => prclib_stack_final
```

⟨*Prclib stacks: procedures*⟩≡
```
    subroutine prclib_stack_final (object)
      class(prclib_stack_t), intent(inout) :: object
      type(prclib_entry_t), pointer :: lib
      do while (associated (object%first))
         lib => object%first
         object%first => lib%next
         call lib%final ()
         deallocate (lib)
      end do
      object%n = 0
    end subroutine prclib_stack_final
```

Output. The entries on the stack will be ordered LIFO, i.e., backwards.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡
```
    procedure :: write => prclib_stack_write
```

⟨Prclib stacks: procedures⟩+≡

```
    subroutine prclib_stack_write (object, unit, libpath)
      class(prclib_stack_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: libpath
      type(prclib_entry_t), pointer :: lib
      integer :: u
      u = output_unit (unit)
      call write_separator_double (u)
      select case (object%n)
      case (0)
        write (u, "(1x,A)")  "Process library stack: [empty]"
      case default
        write (u, "(1x,A)")  "Process library stack:"
        lib => object%first
        do while (associated (lib))
          call write_separator (u)
          call lib%write (u, libpath)
          lib => lib%next
        end do
      end select
      call write_separator_double (u)
    end subroutine prclib_stack_write
```

14.5.3   Operating on Stacks

We take a library entry pointer and push it onto the stack. The previous pointer
is nullified. Subsequently, the library entry is 'owned' by the stack and will be
finalized when the stack is deleted.

⟨Prclib stacks: prclib stack: TBP⟩+≡

```
    procedure :: push => prclib_stack_push
```

⟨Prclib stacks: procedures⟩+≡

```
    subroutine prclib_stack_push (stack, lib)
      class(prclib_stack_t), intent(inout) :: stack
      type(prclib_entry_t), intent(inout), pointer :: lib
      lib%next => stack%first
      stack%first => lib
      lib => null ()
      stack%n = stack%n + 1
    end subroutine prclib_stack_push
```

14.5.4   Accessing Contents

Return a pointer to the topmost stack element. The result type is just the bare
`process_library_t`. There is no `target` attribute required since the stack
elements are allocated via pointers.

⟨Prclib stacks: prclib stack: TBP⟩+≡

```
    procedure :: get_first_ptr => prclib_stack_get_first_ptr
```

⟨*Prclib stacks: procedures*⟩+≡
```
function prclib_stack_get_first_ptr (stack) result (ptr)
  class(prclib_stack_t), intent(in) :: stack
  type(process_library_t), pointer :: ptr
  ptr => stack%first%process_library_t
end function prclib_stack_get_first_ptr
```

Return a complete list of the libraries (names) in the stack. The list is in the order in which the elements got pushed onto the stack, so the 'first' entry is listed last.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡
```
procedure :: get_names => prclib_stack_get_names
```

⟨*Prclib stacks: procedures*⟩+≡
```
subroutine prclib_stack_get_names (stack, libname)
  class(prclib_stack_t), intent(in) :: stack
  type(string_t), dimension(:), allocatable, intent(out) :: libname
  type(prclib_entry_t), pointer :: lib
  integer :: i
  allocate (libname (stack%n))
  i = stack%n
  lib => stack%first
  do while (associated (lib))
     libname(i) = lib%get_name ()
     i = i - 1
     lib => lib%next
  end do
end subroutine prclib_stack_get_names
```

Return a pointer to the library with given name.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡
```
procedure :: get_library_ptr => prclib_stack_get_library_ptr
```

⟨*Prclib stacks: procedures*⟩+≡
```
function prclib_stack_get_library_ptr (stack, libname) result (ptr)
  class(prclib_stack_t), intent(in) :: stack
  type(string_t), intent(in) :: libname
  type(process_library_t), pointer :: ptr
  type(prclib_entry_t), pointer :: current
  current => stack%first
  do while (associated (current))
     if (current%get_name () == libname) then
        ptr => current%process_library_t
        return
     end if
     current => current%next
  end do
  ptr => null ()
end function prclib_stack_get_library_ptr
```

14.5.5   Unit tests

⟨*Prclib stacks: public*⟩+≡

```
    public :: prclib_stacks_test
```

⟨*Prclib stacks: tests*⟩≡
```
    subroutine prclib_stacks_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Prclib stacks: execute tests⟩
    end subroutine prclib_stacks_test
```

Write an empty process library stack

The most trivial test is to write an uninitialized process library stack.

⟨*Prclib stacks: execute tests*⟩≡
```
    call test (prclib_stacks_1, "prclib_stacks_1", &
        "write an empty process library stack", &
        u, results)
```

⟨*Prclib stacks: tests*⟩+≡
```
    subroutine prclib_stacks_1 (u)
      integer, intent(in) :: u
      type(prclib_stack_t) :: stack

      write (u, "(A)")  "* Test output: prclib_stacks_1"
      write (u, "(A)")  "*   Purpose: display an empty process library stack"
      write (u, "(A)")

      call stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_stacks_1"

    end subroutine prclib_stacks_1
```

Fill a process library stack

Fill a process library stack with two (identical) processes.

⟨*Prclib stacks: execute tests*⟩+≡
```
    call test (prclib_stacks_2, "prclib_stacks_2", &
        "fill a process library stack", &
        u, results)
```

⟨*Prclib stacks: tests*⟩+≡
```
    subroutine prclib_stacks_2 (u)
      integer, intent(in) :: u
      type(prclib_stack_t) :: stack
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: prclib_stacks_2"
      write (u, "(A)")  "*   Purpose: fill a process library stack"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize two (empty) libraries &
```

```
         &and push them on the stack"
    write (u, "(A)")

    allocate (lib)
    call lib%init (var_str ("lib1"))
    call stack%push (lib)

    allocate (lib)
    call lib%init (var_str ("lib2"))
    call stack%push (lib)

    call stack%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call stack%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prclib_stacks_2"

  end subroutine prclib_stacks_2
```

14.6   Trivial matrix element for tests

For the purpose of testing the workflow, we implement here two matrix elements
with the simplest possible structure.

   This matrix element generator can only generate a single scattering process
and a single decay process. The scattering process is a quartic interaction of
a massless, neutral and colorless scalar **s** with unit coupling results in a trivial
$2 \to 2$ scattering process. The matrix element is implemented internally, so we
do not need the machinery of external process libraries. The decay process is a
decay of **s** into a pair of colored fermions **f**.

⟨prc_test.f90⟩≡
  ⟨*File header*⟩

```
  module prc_test

    use iso_c_binding !NODEP!
    use kinds !NODEP!
⟨Use strings⟩
    use unit_tests
    use os_interface

    use process_constants
    use prclib_interfaces
    use prc_core_def
    use particle_specifiers
    use process_libraries
```

  ⟨*Standard module head*⟩

⟨*Test ME: public*⟩

⟨*Test ME: types*⟩

```
contains
```

⟨*Test ME: procedures*⟩

⟨*Test ME: tests*⟩

```
end module prc_test
```

14.6.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

⟨*Test ME: public*⟩≡
```
public :: prc_test_def_t
```

⟨*Test ME: types*⟩≡
```
type, extends (prc_core_def_t) :: prc_test_def_t
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in
  type(string_t), dimension(:), allocatable :: prt_out
 contains
 ⟨Test ME: test me def: TBP⟩
end type prc_test_def_t
```

⟨*Test ME: test me def: TBP*⟩≡
```
procedure, nopass :: type_string => prc_test_def_type_string
```

⟨*Test ME: procedures*⟩≡
```
function prc_test_def_type_string () result (string)
  type(string_t) :: string
  string = "test_me"
end function prc_test_def_type_string
```

There is no 'feature' here since there is no external code.

⟨*Test ME: test me def: TBP*⟩+≡
```
procedure, nopass :: get_features => prc_test_def_get_features
```

⟨*Test ME: procedures*⟩+≡
```
subroutine prc_test_def_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (0))
end subroutine prc_test_def_get_features
```

Initialization: set some data (not really useful).

⟨*Test ME: test me def: TBP*⟩+≡
```
procedure :: init => prc_test_def_init
```

1897

⟨*Test ME: procedures*⟩+≡
```
subroutine prc_test_def_init (object, model_name, prt_in, prt_out)
  class(prc_test_def_t), intent(out) :: object
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  object%model_name = model_name
  allocate (object%prt_in (size (prt_in)))
  object%prt_in = prt_in
  allocate (object%prt_out (size (prt_out)))
  object%prt_out = prt_out
end subroutine prc_test_def_init
```

Write/read process- and method-specific data. (No-op)

⟨*Test ME: test me def: TBP*⟩+≡
```
procedure :: write => prc_test_def_write
```

⟨*Test ME: procedures*⟩+≡
```
subroutine prc_test_def_write (object, unit)
  class(prc_test_def_t), intent(in) :: object
  integer, intent(in) :: unit
end subroutine prc_test_def_write
```

⟨*Test ME: test me def: TBP*⟩+≡
```
procedure :: read => prc_test_def_read
```

⟨*Test ME: procedures*⟩+≡
```
subroutine prc_test_def_read (object, unit)
  class(prc_test_def_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine prc_test_def_read
```

Allocate the driver for test ME matrix elements. We get the actual component ID (basename), and we can transfer all process-specific data from the process definition.

⟨*Test ME: test me def: TBP*⟩+≡
```
procedure :: allocate_driver => prc_test_def_allocate_driver
```

⟨*Test ME: procedures*⟩+≡
```
subroutine prc_test_def_allocate_driver (object, driver, basename)
  class(prc_test_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (prc_test_t :: driver)
  select type (driver)
  type is (prc_test_t)
     driver%id = basename
     driver%model_name = object%model_name
     select case (size (object%prt_in))
     case (1);  driver%scattering = .false.
     case (2);  driver%scattering = .true.
     end select
  end select
```

```
      end subroutine prc_test_def_allocate_driver
```

Nothing to connect. This subroutine will not be called.

⟨*Test ME: test me def: TBP*⟩+≡
```
    procedure :: connect => prc_test_def_connect
```

⟨*Test ME: procedures*⟩+≡
```
    subroutine prc_test_def_connect (def, lib_driver, i, proc_driver)
      class(prc_test_def_t), intent(in) :: def
      class(prclib_driver_t), intent(in) :: lib_driver
      integer, intent(in) :: i
      class(prc_core_driver_t), intent(inout) :: proc_driver
    end subroutine prc_test_def_connect
```

14.6.2   Driver

⟨*Test ME: public*⟩+≡
```
    public :: prc_test_t
```

⟨*Test ME: types*⟩+≡
```
    type, extends (process_driver_internal_t) :: prc_test_t
      type(string_t) :: id
      type(string_t) :: model_name
      logical :: scattering = .true.
     contains
     ⟨Test ME: test me driver: TBP⟩
    end type prc_test_t
```

In contrast to generic matrix-element implementations, we can hard-wire the amplitude method as a type-bound procedure.

⟨*Test ME: test me driver: TBP*⟩≡
```
    procedure, nopass :: get_amplitude => prc_test_get_amplitude
```

⟨*Test ME: procedures*⟩+≡
```
    function prc_test_get_amplitude (p) result (amp)
      complex(default) :: amp
      real(default), dimension(:,:), intent(in) :: p
      amp = 1
    end function prc_test_get_amplitude
```

The reported type is the same as for the `prc_test_def_t` type.

⟨*Test ME: test me driver: TBP*⟩+≡
```
    procedure, nopass :: type_name => prc_test_type_name
```

⟨*Test ME: procedures*⟩+≡
```
    function prc_test_type_name () result (string)
      type(string_t) :: string
      string = "test_me"
    end function prc_test_type_name
```

Fill process constants.

⟨*Test ME: test me driver: TBP*⟩+≡
```
    procedure :: fill_constants => prc_test_fill_constants
```

⟨Test ME: procedures⟩+≡

```fortran
subroutine prc_test_fill_constants (driver, data)
  class(prc_test_t), intent(in) :: driver
  type(process_constants_t), intent(out) :: data
  data%id = driver%id
  data%model_name = driver%model_name
  if (driver%scattering) then
     data%n_in  = 2
     data%n_out = 2
     data%n_flv = 1
     data%n_hel = 1
     data%n_col = 1
     data%n_cin = 2
     data%n_cf  = 1
     allocate (data%flv_state (4, 1))
     data%flv_state = 25
     allocate (data%hel_state (4, 1))
     data%hel_state = 0
     allocate (data%col_state (2, 4, 1))
     data%col_state = 0
     allocate (data%ghost_flag (4, 1))
     data%ghost_flag = .false.
     allocate (data%color_factors (1))
     data%color_factors = 1
     allocate (data%cf_index (2, 1))
     data%cf_index = 1
  else
     data%n_in  = 1
     data%n_out = 2
     data%n_flv = 1
     data%n_hel = 2
     data%n_col = 1
     data%n_cin = 2
     data%n_cf  = 1
     allocate (data%flv_state (3, 1))
     data%flv_state(:,1) = [25, 6, -6]
     allocate (data%hel_state (3, 2))
     data%hel_state(:,1) = [0, 1,-1]
     data%hel_state(:,2) = [0,-1, 1]
     allocate (data%col_state (2, 3, 1))
     data%col_state = reshape ([0,0, 1,0, 0,-1], [2,3,1])
     allocate (data%ghost_flag (3, 1))
     data%ghost_flag = .false.
     allocate (data%color_factors (1))
     data%color_factors = 3
     allocate (data%cf_index (2, 1))
     data%cf_index = 1
  end if
end subroutine prc_test_fill_constants
```

14.6.3 Shortcut

Since this module is there for testing purposes, we set up a subroutine that does all the work at once: create a library with the two processes (scattering and decay), configure and load, and set up the driver.

⟨*Test ME: public*⟩+≡
```
    public :: prc_test_create_library
```

⟨*Test ME: procedures*⟩+≡
```
    subroutine prc_test_create_library &
        (libname, lib, scattering, decay, procname1, procname2)
      type(string_t), intent(in) :: libname
      type(process_library_t), intent(out) :: lib
      logical, intent(in), optional :: scattering, decay
      type(string_t), intent(in), optional :: procname1, procname2
      type(string_t) :: model_name, procname
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      class(prc_core_def_t), allocatable :: def
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      logical :: sca, dec
      sca = .true.;   if (present (scattering))  sca = scattering
      dec = .false.;  if (present (decay))       dec = decay

      call os_data_init (os_data)
      call lib%init (libname)
      model_name = "Test"

      if (sca) then
         if (present (procname1)) then
            procname = procname1
         else
            procname = libname
         end if
         allocate (prt_in (2), prt_out (2))
         prt_in  = [var_str ("s"), var_str ("s")]
         prt_out = [var_str ("s"), var_str ("s")]
         allocate (prc_test_def_t :: def)
         select type (def)
         type is (prc_test_def_t)
            call def%init (model_name, prt_in, prt_out)
         end select
         allocate (entry)
         call entry%init (procname, model_name = model_name, &
              n_in = 2, n_components = 1)
         call entry%import_component (1, n_out = size (prt_out), &
              prt_in  = new_prt_spec (prt_in), &
              prt_out = new_prt_spec (prt_out), &
              method  = var_str ("test_me"), &
              variant = def)
         call lib%append (entry)
      end if

      if (dec) then
         if (present (procname2)) then
```

```
         procname = procname2
      else
         procname = libname
      end if
      if (allocated (prt_in))  deallocate (prt_in, prt_out)
      allocate (prt_in (1), prt_out (2))
      prt_in  = [var_str ("s")]
      prt_out = [var_str ("f"), var_str ("fbar")]
      allocate (prc_test_def_t :: def)
      select type (def)
      type is (prc_test_def_t)
         call def%init (model_name, prt_in, prt_out)
      end select
      allocate (entry)
      call entry%init (procname, model_name = model_name, &
           n_in = 1, n_components = 1)
      call entry%import_component (1, n_out = size (prt_out), &
           prt_in  = new_prt_spec (prt_in), &
           prt_out = new_prt_spec (prt_out), &
           method  = var_str ("test_decay"), &
           variant = def)
      call lib%append (entry)
    end if

    call lib%configure (os_data)
    call lib%load (os_data)
  end subroutine prc_test_create_library
```

14.6.4   Test

This is the master for calling self-test procedures.

⟨*Test ME: public*⟩+≡
```
    public :: prc_test_test
```

⟨*Test ME: tests*⟩≡
```
    subroutine prc_test_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Test ME: execute tests⟩
  end subroutine prc_test_test
```

Generate and load the scattering process

The process is $ss \to ss$, where s is a trivial scalar particle, for vanishing mass and unit coupling. We initialize the process, build the library, and compute the particular matrix element for momenta of unit energy and right-angle scattering. (The scattering is independent of angle.) The matrix element is equal to unity.

⟨*Test ME: execute tests*⟩≡
```
    call test (prc_test_1, "prc_test_1", &
        "build and load trivial process", &
        u, results)
```

⟨Test ME: tests⟩+≡

```
subroutine prc_test_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: driver
  real(default), dimension(0:3,4) :: p
  integer :: i

  write (u, "(A)")  "* Test output: prc_test_1"
  write (u, "(A)")  "*   Purpose: create a trivial process"
  write (u, "(A)")  "*            build a library and &
       &access the matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call os_data_init (os_data)
  call lib%init (var_str ("prc_test1"))

  model_name = "Test"
  allocate (prt_in (2), prt_out (2))
  prt_in  = [var_str ("s"), var_str ("s")]
  prt_out = [var_str ("s"), var_str ("s")]

  allocate (prc_test_def_t :: def)
  select type (def)
  type is (prc_test_def_t)
     call def%init (model_name, prt_in, prt_out)
  end select
  allocate (entry)
  call entry%init (var_str ("prc_test1_a"), model_name = model_name, &
       n_in = 2, n_components = 1)
  call entry%import_component (1, n_out = size (prt_out), &
       prt_in  = new_prt_spec (prt_in), &
       prt_out = new_prt_spec (prt_out), &
       method  = var_str ("test_me"), &
       variant = def)
  call lib%append (entry)

  write (u, "(A)")  "* Configure library"
  write (u, "(A)")
  call lib%configure (os_data)

  write (u, "(A)")  "* Load library"
  write (u, "(A)")
  call lib%load (os_data)

  call lib%write (u)
```

1903

```
write (u, "(A)")
write (u, "(A)")   "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")   "is active                  = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")   "n_processes                = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")   "* Constants of prc_test1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("prc_test1_a"), 1, data, driver)

write (u, "(1x,A,A)")   "component ID    = ", char (data%id)
write (u, "(1x,A,A)")   "model name      = ", char (data%model_name)
write (u, "(1x,A,A,A)")   "md5sum          = '", data%md5sum, "'"
write (u, "(1x,A,L1)")   "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)")   "n_in  = ", data%n_in
write (u, "(1x,A,I0)")   "n_out = ", data%n_out
write (u, "(1x,A,I0)")   "n_flv = ", data%n_flv
write (u, "(1x,A,I0)")   "n_hel = ", data%n_hel
write (u, "(1x,A,I0)")   "n_col = ", data%n_col
write (u, "(1x,A,I0)")   "n_cin = ", data%n_cin
write (u, "(1x,A,I0)")   "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))")   "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))")   "hel state =", data%hel_state(:,1)
write (u, "(1x,A,10(1x,I0))")   "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))")   "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))")   "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))")   "cf index =", data%cf_index

write (u, "(A)")
write (u, "(A)")   "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_default, 0.0_default, 0.0_default, 1.0_default, &
     1.0_default, 0.0_default, 0.0_default,-1.0_default, &
     1.0_default, 1.0_default, 0.0_default, 0.0_default, &
     1.0_default,-1.0_default, 0.0_default, 0.0_default &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")   "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")   "* Compute matrix element:"
write (u, "(A)")

select type (driver)
type is (prc_test_t)
```

```
      write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
    end select

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_test_1"

  end subroutine prc_test_1
```

Shortcut

This is identical to the previous test, but we create the library be a single command. This is handy for other modules which use the test process.

⟨*Test ME: execute tests*⟩+≡
```
    call test (prc_test_2, "prc_test_2", &
         "build and load trivial process using shortcut", &
         u, results)
```

⟨*Test ME: tests*⟩+≡
```
  subroutine prc_test_2 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    class(prc_core_driver_t), allocatable :: driver
    type(process_constants_t) :: data
    real(default), dimension(0:3,4) :: p

    write (u, "(A)")  "* Test output: prc_test_2"
    write (u, "(A)")  "*   Purpose: create a trivial process"
    write (u, "(A)")  "*            build a library and &
         &access the matrix element"
    write (u, "(A)")

    write (u, "(A)")  "* Build and load a process library with one entry"

    call prc_test_create_library (var_str ("prc_test2"), lib)
    call lib%connect_process (var_str ("prc_test2"), 1, data, driver)

    p = reshape ([ &
         1.0_default, 0.0_default, 0.0_default, 1.0_default, &
         1.0_default, 0.0_default, 0.0_default,-1.0_default, &
         1.0_default, 1.0_default, 0.0_default, 0.0_default, &
         1.0_default,-1.0_default, 0.0_default, 0.0_default &
         ], [4,4])

    write (u, "(A)")
    write (u, "(A)")  "* Compute matrix element:"
    write (u, "(A)")

    select type (driver)
    type is (prc_test_t)
       write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
    end select
```

```
    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_test_2"

  end subroutine prc_test_2
```

Generate and load the decay process

The process is $s \to f\bar{f}$, where s is a trivial scalar particle and f is a colored fermion. We initialize the process, build the library, and compute the particular matrix element for a fixed momentum configuration. (The decay is independent of angle.) The matrix element is equal to unity.

⟨*Test ME: execute tests*⟩+≡
```
    call test (prc_test_3, "prc_test_3", &
         "build and load trivial decay", &
         u, results)
```

⟨*Test ME: tests*⟩+≡
```
    subroutine prc_test_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(process_library_t) :: lib
      class(prc_core_def_t), allocatable :: def
      type(process_def_entry_t), pointer :: entry
      type(string_t) :: model_name
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(process_constants_t) :: data
      class(prc_core_driver_t), allocatable :: driver
      real(default), dimension(0:3,3) :: p
      integer :: i

      write (u, "(A)")  "* Test output: prc_test_3"
      write (u, "(A)")  "*   Purpose: create a trivial decay process"
      write (u, "(A)")  "*            build a library and &
           &access the matrix element"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize a process library with one entry"
      write (u, "(A)")
      call os_data_init (os_data)
      call lib%init (var_str ("prc_test3"))

      model_name = "Test"
      allocate (prt_in (2), prt_out (2))
      prt_in  = [var_str ("s")]
      prt_out = [var_str ("f"), var_str ("F")]

      allocate (prc_test_def_t :: def)
      select type (def)
      type is (prc_test_def_t)
         call def%init (model_name, prt_in, prt_out)
```

```
end select
allocate (entry)
call entry%init (var_str ("prc_test3_a"), model_name = model_name, &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("test_me"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Load library"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of prc_test3_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("prc_test3_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,L1)")  "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)")  "n_in  = ", data%n_in
write (u, "(1x,A,I0)")  "n_out = ", data%n_out
write (u, "(1x,A,I0)")  "n_flv = ", data%n_flv
write (u, "(1x,A,I0)")  "n_hel = ", data%n_hel
write (u, "(1x,A,I0)")  "n_col = ", data%n_col
write (u, "(1x,A,I0)")  "n_cin = ", data%n_cin
write (u, "(1x,A,I0)")  "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))")  "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))")  "hel state =", data%hel_state(:,1)
write (u, "(1x,A,10(1x,I2))")  "hel state =", data%hel_state(:,2)
write (u, "(1x,A,10(1x,I0))")  "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))")  "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))")  "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))")  "cf index =", data%cf_index
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Set kinematics:"
      write (u, "(A)")

      p = reshape ([ &
            125._default, 0.0_default, 0.0_default, 0.0_default, &
            62.5_default, 0.0_default, 0.0_default, 62.5_default, &
            62.5_default, 0.0_default, 0.0_default,-62.5_default &
            ], [4,3])
      do i = 1, 3
         write (u, "(2x,A,I0,A,4(1x,F8.4))")  "p", i, " =", p(:,i)
      end do

      write (u, "(A)")
      write (u, "(A)")   "* Compute matrix element:"
      write (u, "(A)")

      select type (driver)
      type is (prc_test_t)
         write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
      end select

      call lib%final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: prc_test_3"

    end subroutine prc_test_3
```

Shortcut

This is identical to the previous test, but we create the library be a single command. This is handy for other modules which use the test process.

⟨*Test ME: execute tests*⟩+≡
```
      call test (prc_test_4, "prc_test_4", &
            "build and load trivial decay using shortcut", &
            u, results)
```

⟨*Test ME: tests*⟩+≡
```
    subroutine prc_test_4 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      class(prc_core_driver_t), allocatable :: driver
      type(process_constants_t) :: data
      real(default), dimension(0:3,3) :: p

      write (u, "(A)")  "* Test output: prc_test_4"
      write (u, "(A)")  "*   Purpose: create a trivial decay process"
      write (u, "(A)")  "*            build a library and &
            &access the matrix element"
      write (u, "(A)")
```

```
      write (u, "(A)")  "* Build and load a process library with one entry"

      call prc_test_create_library (var_str ("prc_test4"), lib, &
           scattering=.false., decay=.true.)
      call lib%connect_process (var_str ("prc_test4"), 1, data, driver)

   p = reshape ([ &
        125._default, 0.0_default, 0.0_default, 0.0_default, &
        62.5_default, 0.0_default, 0.0_default, 62.5_default, &
        62.5_default, 0.0_default, 0.0_default,-62.5_default &
        ], [4,3])

   write (u, "(A)")
   write (u, "(A)")  "* Compute matrix element:"
   write (u, "(A)")

   select type (driver)
   type is (prc_test_t)
      write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
   end select

   call lib%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: prc_test_4"

end subroutine prc_test_4
```

Chapter 15

Integration and Process Objects

This is the central part of the WHIZARD package. It provides the functionality for evaluating structure functions, kinematics and matrix elements, integration and event generation. It combines the various parts that deal with those tasks individually and organizes the data transfer between them.

integration_results This module handles results from integrating processes. It records passes and iterations, calculates statistical averages, and provides the user output of integration results.

prc_core Here, we define the abstract `prc_core_t` type which handles all specific features of kinematics matrix-element evaluation that depend on a particular class of processes.

parton_states A `parton_state_t` object represents an elementary partonic interaction. There are two versions: one for the isolated elementary process, one for the elementary process convoluted with the structure-function chain. The parton state is an effective state. It needs not coincide with the seed-kinematics state which is used in evaluating phase space.

processes Here, all pieces are combined for the purpose of evaluating the elementary processes. The whole algorithm is coded in terms of abstract data types as defined in the appropriate modules: `prc_core` for matrix-element evaluation, `prc_core_def` for the associated configuration and driver, `sf_base` for beams and structure-functions, `phs_base` for phase space, and `mci_base` for integration and event generation.

process_stacks Process stacks collect process objects.

15.1 Integration results

We record integration results and errors in a dedicated type. This allows us to do further statistics such as weighted average, chi-squared, grouping by integration passes, etc.

Note WHIZARD 2.2.0: This code is taken from the previous `processes` module essentially unchanged and converted into a separate module. It lacks an overhaul and, in particular, self-tests.

⟨integration_results.f90⟩≡

```
module integration_results

  use limits, only: FMT_12, FMT_14, GML_MIN_RANGE_RATIO !NODEP!
  use limits, only: INTEGRATION_ERROR_TOLERANCE !NODEP!
```
⟨*Use kinds*⟩
⟨*Use strings*⟩
⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use md5
  use os_interface
  use mci_base
```

⟨*Standard module head*⟩

⟨*Integration results: public*⟩

⟨*Integration results: parameters*⟩

⟨*Integration results: types*⟩

```
contains
```

⟨*Integration results: procedures*⟩

```
end module integration_results
```

15.1.1 Integration results entry

This object collects the results of an integration pass and makes them available to the outside.

The results object has to distinguish the process type:

We store the process type, the index of the integration pass and the absolute iteration index, the number of iterations contained in this result (for averages), and the integral (cross section or partial width), error estimate, efficiency.

For intermediate results, we set a flag if this result is an improvement w.r.t. previous ones.

The process type indicates decay or scattering. Dummy entries (skipped iterations) have a process type of `PRC_UNKNOWN`.

⟨*Integration results: types*⟩≡

```
    type :: integration_entry_t
      private
      integer :: process_type = PRC_UNKNOWN
      integer :: pass = 0
      integer :: it = 0
      integer :: n_it = 0
      integer :: n_calls = 0
      logical :: improved = .false.
      real(default) :: integral = 0
```

```
    real(default) :: error = 0
    real(default) :: efficiency = 0
    real(default) :: chi2 = 0
    real(default), dimension(:), allocatable :: chain_weights
  end type integration_entry_t
```

The possible values of the type indicator:

⟨*Integration results: parameters*⟩≡
```
    integer, parameter, public :: PRC_UNKNOWN = 0
    integer, parameter, public :: PRC_DECAY = 1
    integer, parameter, public :: PRC_SCATTERING = 2
```

Initialize with all relevant data

⟨*Integration results: procedures*⟩≡
```
    subroutine integration_entry_init (entry, &
         process_type, pass, it, n_it, n_calls, improved, &
         integral, error, efficiency, chi2, chain_weights)
      type(integration_entry_t), intent(out) :: entry
      integer, intent(in) :: process_type, pass, it, n_it, n_calls
      logical, intent(in) :: improved
      real(default), intent(in) :: integral, error, efficiency
      real(default), intent(in), optional :: chi2
      real(default), dimension(:), intent(in), optional :: chain_weights
      entry%process_type = process_type
      entry%pass = pass
      entry%it = it
      entry%n_it = n_it
      entry%n_calls = n_calls
      entry%improved = improved
      entry%integral = integral
      entry%error = error
      entry%efficiency = efficiency
      if (present (chi2)) &
           entry%chi2 = chi2
      if (present (chain_weights)) then
         allocate (entry%chain_weights (size (chain_weights)))
         entry%chain_weights = chain_weights
      end if
    end subroutine integration_entry_init
```

Access values, some of them computed on demand:

⟨*Integration results: procedures*⟩+≡
```
    elemental function integration_entry_get_pass (entry) result (n)
      integer :: n
      type(integration_entry_t), intent(in) :: entry
      n = entry%pass
    end function integration_entry_get_pass

    elemental function integration_entry_get_n_calls (entry) result (n)
      integer :: n
      type(integration_entry_t), intent(in) :: entry
      n = entry%n_calls
```

```
      end function integration_entry_get_n_calls

  elemental function integration_entry_get_integral (entry) result (int)
    real(default) :: int
    type(integration_entry_t), intent(in) :: entry
    int = entry%integral
  end function integration_entry_get_integral

  elemental function integration_entry_get_error (entry) result (err)
    real(default) :: err
    type(integration_entry_t), intent(in) :: entry
    err = entry%error
  end function integration_entry_get_error

  elemental function integration_entry_get_relative_error (entry) result (err)
    real(default) :: err
    type(integration_entry_t), intent(in) :: entry
    if (entry%integral /= 0) then
       err = entry%error / entry%integral
    else
       err = 0
    end if
  end function integration_entry_get_relative_error

  elemental function integration_entry_get_accuracy (entry) result (acc)
    real(default) :: acc
    type(integration_entry_t), intent(in) :: entry
    acc = accuracy (entry%integral, entry%error, entry%n_calls)
  end function integration_entry_get_accuracy

  elemental function accuracy (integral, error, n_calls) result (acc)
    real(default) :: acc
    real(default), intent(in) :: integral, error
    integer, intent(in) :: n_calls
    if (integral /= 0) then
       acc = error / integral * sqrt (real (n_calls, default))
    else
       acc = 0
    end if
  end function accuracy

  elemental function integration_entry_get_efficiency (entry) result (eff)
    real(default) :: eff
    type(integration_entry_t), intent(in) :: entry
    eff = entry%efficiency
  end function integration_entry_get_efficiency

  elemental function integration_entry_get_chi2 (entry) result (chi2)
    real(default) :: chi2
    type(integration_entry_t), intent(in) :: entry
    chi2 = entry%chi2
  end function integration_entry_get_chi2

  elemental function integration_entry_has_improved (entry) result (flag)
```

```
      logical :: flag
      type(integration_entry_t), intent(in) :: entry
      flag = entry%improved
    end function integration_entry_has_improved

    elemental function integration_entry_get_n_groves (entry) result (n_groves)
      integer :: n_groves
      type(integration_entry_t), intent(in) :: entry
      if (allocated (entry%chain_weights)) then
         n_groves = size (entry%chain_weights, 1)
      else
         n_groves = 0
      end if
    end function integration_entry_get_n_groves
```

Output. This writes the header line for the result account below:

⟨*Integration results: procedures*⟩+≡
```
    subroutine write_header (process_type, unit, logfile)
      integer, intent(in) :: process_type
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: logfile
      character(5) :: phys_unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      select case (process_type)
      case (PRC_DECAY);       phys_unit = "[GeV]"
      case (PRC_SCATTERING); phys_unit = "[fb] "
      case default
         phys_unit = ""
      end select
      write (msg_buffer, "(A)") &
           "It      Calls  Integral" // phys_unit // &
           " Error" // phys_unit // &
           "  Err[%]    Acc  Eff[%]   Chi2 N[It] |"
      call msg_message (unit=u, logfile=logfile)
    end subroutine write_header
```

This writes a separator for result display:

⟨*Integration results: procedures*⟩+≡
```
    subroutine write_hline (unit)
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "|" // (repeat ("-", 77)) // "|"
      flush (u)
    end subroutine write_hline

    subroutine write_dline (unit)
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "|" // (repeat ("=", 77)) // "|"
      flush (u)
```

1914

```
      end subroutine write_dline
```

This writes the standard result account into one screen line. The verbose version
uses multiple lines and prints the unabridged values. Dummy entries are not
written.

⟨*Integration results: procedures*⟩+≡

```
    subroutine integration_entry_write (entry, unit, verbose, suppress)
      type(integration_entry_t), intent(in) :: entry
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      logical, intent(in), optional :: suppress
      integer :: u
      character(1) :: star
      character(12) :: fmt
      logical :: verb, supp
      u = output_unit (unit);  if (u < 0)  return
      verb = .false.;  if (present (verbose))  verb = verbose
      supp = .false.;  if (present (suppress)) supp = suppress
      if (verb)  then
         write (u, *)  "process_type = ", entry%process_type
         write (u, *)  "        pass = ", entry%pass
         write (u, *)  "          it = ", entry%it
         write (u, *)  "        n_it = ", entry%n_it
         write (u, *)  "     n_calls = ", entry%n_calls
         write (u, *)  "    improved = ", entry%improved
         write (u, *)  "    integral = ", entry%integral
         write (u, *)  "       error = ", entry%error
         write (u, *)  "  efficiency = ", entry%efficiency
         write (u, *)  "        chi2 = ", entry%chi2
         if (allocated (entry%chain_weights)) then
            write (u, *)  "    n_groves = ", size (entry%chain_weights)
            write (u, *)  "chain_weights = ", entry%chain_weights
         else
            write (u, *)  "    n_groves = 0"
         end if
      else if (entry%process_type /= PRC_UNKNOWN) then
         if (entry%improved .and. .not. supp) then
            star = "*"
         else
            star = " "
         end if
         call pac_fmt (fmt, FMT_14, "1x," // FMT_12 // ",1x", suppress)
         if (entry%n_it /= 1) then
            write (u, "(1x,I3,1x,I10,1x," // fmt // ",1x,ES9.2,1x,F7.2," // &
              "1x,F7.2,A1,1x,F6.2,1x,F7.2,1x,I3)") &
                 entry%it, &
                 entry%n_calls, &
                 entry%integral, &
                 abs(entry%error), &
                 abs(integration_entry_get_relative_error (entry)) * 100, &
                 abs(integration_entry_get_accuracy (entry)), &
                 star, &
                 entry%efficiency * 100, &
```

```
                       entry%chi2, &
                       entry%n_it
               else
                   write (u, "(1x,I3,1x,I10,1x," // fmt // ",1x,ES9.2,1x,F7.2," // &
                   "1x,F7.2,A1,1x,F6.2,1x,F7.2,1x,I3)") &
                       entry%it, &
                       entry%n_calls, &
                       entry%integral, &
                       abs(entry%error), &
                       abs(integration_entry_get_relative_error (entry)) * 100, &
                       abs(integration_entry_get_accuracy (entry)), &
                       star, &
                       entry%efficiency * 100
               end if
           end if
           flush (u)
       end subroutine integration_entry_write
```

Read the entry, assuming it has been written in verbose format.

⟨*Integration results: procedures*⟩+≡
```
       subroutine integration_entry_read (entry, unit)
         type(integration_entry_t), intent(out) :: entry
         integer, intent(in) :: unit
         character(30) :: dummy
         character :: equals
         integer :: n_groves
         read (unit, *)  dummy, equals, entry%process_type
         read (unit, *)  dummy, equals, entry%pass
         read (unit, *)  dummy, equals, entry%it
         read (unit, *)  dummy, equals, entry%n_it
         read (unit, *)  dummy, equals, entry%n_calls
         read (unit, *)  dummy, equals, entry%improved
         read (unit, *)  dummy, equals, entry%integral
         read (unit, *)  dummy, equals, entry%error
         read (unit, *)  dummy, equals, entry%efficiency
         read (unit, *)  dummy, equals, entry%chi2
         read (unit, *)  dummy, equals, n_groves
         if (n_groves /= 0) then
            allocate (entry%chain_weights (n_groves))
            read (unit, *)  dummy, equals, entry%chain_weights
         end if
       end subroutine integration_entry_read
```

Write an account of the channel weights, accumulated by groves.

⟨*Integration results: procedures*⟩+≡
```
       subroutine integration_entry_write_chain_weights (entry, unit)
         type(integration_entry_t), intent(in) :: entry
         integer, intent(in), optional :: unit
         integer :: u, i
         u = output_unit (unit);  if (u < 0)  return
         if (allocated (entry%chain_weights)) then
            do i = 1, size (entry%chain_weights)
               write (u, "(1x,I3)", advance="no")  nint (entry%chain_weights(i) * 100)
```

```
      end do
      write (u, *)
    end if
  end subroutine integration_entry_write_chain_weights
```

Compute the average for all entries in the specified integration pass. The integrals are weighted w.r.t. their individual errors.

The quoted error of the result is the expected error, computed from the weighted average of the given individual errors.

This should be compared to the actual distribution of the results, from which we also can compute an error estimate if there is more than one iteration. The ratio of the distribution error and the averaged error, is the χ^2 value.

All error distributions are assumed Gaussian, of course. The χ^2 value is a partial check for this assumption. If it is significantly greater than unity, there is something wrong with the individual errors.

The efficiency returned is the one of the last entry in the integration pass.

If any error vanishes, averaging by this algorithm would fail. In this case, we simply average the entries and use the deviations from this average (if any) to estimate the error.

⟨*Integration results: procedures*⟩+≡
```
  function compute_average (entry, pass) result (result)
    type(integration_entry_t) :: result
    type(integration_entry_t), dimension(:), intent(in) :: entry
    integer, intent(in) :: pass
    integer :: i
    logical, dimension(size(entry)) :: mask
    real(default), dimension(size(entry)) :: ivar
    real(default) :: sum_ivar, variance
    result%process_type = entry(1)%process_type
    result%pass = pass
    mask = entry%pass == pass .and. entry%process_type /= PRC_UNKNOWN
    result%it = maxval (entry%it, mask)
    result%n_it = count (mask)
    result%n_calls = sum (entry%n_calls, mask)
    if (.not. any (mask .and. entry%error == 0)) then
       where (mask)
          ivar = 1 / entry%error ** 2
       elsewhere
          ivar = 0
       end where
       sum_ivar = sum (ivar, mask)
       if (sum_ivar /= 0) then
          variance = 1 / sum_ivar
       else
          variance = 0
       end if
       result%integral = sum (entry%integral * ivar, mask) * variance
       if (result%n_it > 1) then
          result%chi2 = &
               sum ((entry%integral - result%integral)**2 * ivar, mask) &
               / (result%n_it - 1)
       end if
```

1917

```
      else if (result%n_it /= 0) then
         result%integral = sum (entry%integral, mask) / result%n_it
         if (result%n_it > 1) then
            variance = &
                 sum ((entry%integral - result%integral)**2, mask) &
                 / (result%n_it - 1)
            if (result%integral /= 0) then
               if (abs (variance / result%integral) &
                    < 100 * epsilon (1._default)) then
                  variance = 0
               end if
            end if
            result%chi2 = variance / result%n_it
         else
            variance = 0
         end if
      end if
      result%error = sqrt (variance)
      do i = size (entry), 1, -1
         if (mask(i)) then
            result%efficiency = entry(i)%efficiency
            exit
         end if
      end do
   end function compute_average
```

15.1.2   Combined integration results

We collect a list of results which grows during the execution of the program. This is implemented as an array which grows if necessary; so we can easily compute averages.

   We implement this as an extension of the `mci_results_t` which is defined in `mci_base` as an abstract type. We thus decouple the implementation of the integrator from the implementation of the results display, but nevertheless can record intermediate results during integration. This implies that the present extension implements a `record` method.

⟨*Integration results: public*⟩≡
```
    public :: integration_results_t
```

⟨*Integration results: types*⟩+≡
```
    type, extends (mci_results_t) :: integration_results_t
      private
      integer :: process_type = PRC_UNKNOWN
      integer :: current_pass = 0
      integer :: n_pass = 0
      integer :: n_it = 0
      logical :: screen = .false.
      integer :: unit = 0
      real(default) :: error_threshold = 0
      type(integration_entry_t), dimension(:), allocatable :: entry
      type(integration_entry_t), dimension(:), allocatable :: average
   contains
```

⟨*Integration results: integration results: TBP*⟩
```
    end type integration_results_t
```

The array is extended in chunks of 10 entries.

⟨*Integration results: parameters*⟩+≡
```
    integer, parameter :: RESULTS_CHUNK_SIZE = 10
```

The standard does not require to explicitly initialize the integers; however, some gfortran version has a bug here and misses the default initialization in the type definition.

⟨*Integration results: integration results: TBP*⟩≡
```
    procedure :: init => integration_results_init
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_init (results, process_type)
      class(integration_results_t), intent(out) :: results
      integer, intent(in) :: process_type
      results%process_type = process_type
      results%n_pass = 0
      results%n_it = 0
      allocate (results%entry (RESULTS_CHUNK_SIZE))
      allocate (results%average (RESULTS_CHUNK_SIZE))
    end subroutine integration_results_init
```

Set addititional parameters: the `error_threshold` declares that any error value (in absolute numbers) smaller than this is to be considered zero.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: set_error_threshold => integration_results_set_error_threshold
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_set_error_threshold (results, error_threshold)
      class(integration_results_t), intent(inout) :: results
      real(default), intent(in) :: error_threshold
      results%error_threshold = error_threshold
    end subroutine integration_results_set_error_threshold
```

Output (ASCII format). The `verbose` format is used for writing the header in grid files.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: write => integration_results_write
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_write (object, unit, verbose, suppress)
      class(integration_results_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      logical, intent(in), optional :: suppress
      logical :: verb
      integer :: u, n
      u = output_unit (unit);  if (u < 0)  return
      verb = .false.;  if (present (verbose))  verb = verbose
      if (.not. verb) then
         call write_dline (unit)
```

```
      if (object%n_it /= 0) then
          call write_header (object%entry(1)%process_type, unit, &
               logfile=.false.)
          call write_dline (unit)
          do n = 1, object%n_it
              if (n > 1) then
                  if (object%entry(n)%pass /= object%entry(n-1)%pass) then
                      call write_hline (unit)
                      call integration_entry_write &
                           (object%average(object%entry(n-1)%pass), &
                                unit, suppress = suppress)
                      call write_hline (unit)
                  end if
              end if
              call integration_entry_write (object%entry(n), unit, &
                    suppress = suppress)
          end do
          call write_hline(unit)
          call integration_entry_write (object%average(object%n_pass), &
                unit, suppress = suppress)
      else
          call msg_message ("[WHIZARD integration results: empty]", unit)
      end if
      call write_dline (unit)
  else
      write (u, *)  "begin(integration_results)"
      write (u, *)  "  n_pass = ", object%n_pass
      write (u, *)  "    n_it = ", object%n_it
      if (object%n_it > 0) then
          write (u, *)  "begin(integration_pass)"
          do n = 1, object%n_it
              if (n > 1) then
                  if (object%entry(n)%pass /= object%entry(n-1)%pass) then
                      write (u, *)  "end(integration_pass)"
                      write (u, *)  "begin(integration_pass)"
                  end if
              end if
              write (u, *)  "begin(iteration)"
              call integration_entry_write (object%entry(n), unit, &
                    verbose = verb, suppress = suppress)
              write (u, *)  "end(iteration)"
          end do
          write (u, *)  "end(integration_pass)"
      end if
      write (u, *)  "end(integration_results)"
  end if
  flush (u)
end subroutine integration_results_write
```

During integration, we do not want to print all results at once, but each inter-
mediate result as soon as we get it. Thus, the previous procedure is chopped in
pieces. First piece: store the output unit and a flag whether we want to print
to standard output as well. Then write the header if the results are still empty,

i.e., before integration has started. The second piece writes a single result to the saved output channels. We call this from the `record` method, which can be called from the integrator directly. The third piece writes the average result, once a pass has been completed. The fourth piece writes a footer (if any), assuming that this is the final result.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: display_init => integration_results_display_init
procedure :: display_current => integration_results_display_current
procedure :: display_pass => integration_results_display_pass
procedure :: display_final => integration_results_display_final
```

⟨*Integration results: procedures*⟩+≡
```
subroutine integration_results_display_init &
    (results, process_type, screen, unit)
  class(integration_results_t), intent(inout) :: results
  integer, intent(in) :: process_type
  logical, intent(in) :: screen
  integer, intent(in), optional :: unit
  integer :: u
  if (present (unit))  results%unit = unit
  u = output_unit ()
  results%screen = screen
  if (results%n_it == 0) then
     if (results%screen) then
        call write_dline (u)
        call write_header (process_type, u, &
             logfile=.false.)
        call write_dline (u)
     end if
     if (results%unit /= 0) then
        call write_dline (results%unit)
        call write_header (process_type, results%unit, &
             logfile=.false.)
        call write_dline (results%unit)
     end if
  else
     if (results%screen) then
        call write_hline (u)
     end if
     if (results%unit /= 0) then
        call write_hline (results%unit)
     end if
  end if
end subroutine integration_results_display_init

subroutine integration_results_display_current (results, pacify)
  class(integration_results_t), intent(in) :: results
  integer :: u
  logical, intent(in), optional :: pacify
  u = output_unit ()
  if (results%screen) then
     call integration_entry_write (results%entry(results%n_it), u, &
          suppress = pacify)
  end if
```

```
      if (results%unit /= 0) then
         call integration_entry_write (results%entry(results%n_it), &
              results%unit, suppress = pacify)
      end if
    end subroutine integration_results_display_current

    subroutine integration_results_display_pass (results, pacify)
      class(integration_results_t), intent(in) :: results
      logical, intent(in), optional :: pacify
      integer :: u
      u = output_unit ()
      if (results%screen) then
         call write_hline (u)
         call integration_entry_write &
              (results%average(results%entry(results%n_it)%pass), &
                 u, suppress = pacify)
      end if
      if (results%unit /= 0) then
         call write_hline (results%unit)
         call integration_entry_write &
              (results%average(results%entry(results%n_it)%pass), &
                 results%unit, suppress = pacify)
      end if
    end subroutine integration_results_display_pass

    subroutine integration_results_display_final (results)
      class(integration_results_t), intent(inout) :: results
      integer :: u
      u = output_unit ()
      if (results%screen) then
         call write_dline (u)
      end if
      if (results%unit /= 0) then
         call write_dline (results%unit)
      end if
      results%screen = .false.
      results%unit = 0
    end subroutine integration_results_display_final
```

Write a concise table of chain weights, i.e., the channel history where channels are collected by chains.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: write_chain_weights => &
        integration_results_write_chain_weights
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_write_chain_weights (results, unit)
      class(integration_results_t), intent(in) :: results
      integer, intent(in), optional :: unit
      integer :: u, i, n
      u = output_unit (unit);  if (u < 0)  return
      if (allocated (results%entry(1)%chain_weights) .and. results%n_it /= 0) then
         call msg_message ("Phase-space chain (grove) weight history: " &
              // "(numbers in %)", unit)
```

```
              write (u, "(A9)", advance="no")  "| chain |"
                 do i = 1, integration_entry_get_n_groves (results%entry(1))
                 write (u, "(1x,I3)", advance="no")  i
              end do
              write (u, *)
              call write_dline (unit)
              do n = 1, results%n_it
                 if (n > 1) then
                    if (results%entry(n)%pass /= results%entry(n-1)%pass) then
                       call write_hline (unit)
                    end if
                 end if
                 write (u, "(1x,I6,1x,A1)", advance="no")  n, "|"
                 call integration_entry_write_chain_weights (results%entry(n), unit)
              end do
              flush (u)
              call write_dline(unit)
           end if
        end subroutine integration_results_write_chain_weights
```

Read the list from file. The file must be written using the **verbose** option of the writing routine.

⟨*Integration results: procedures*⟩+≡

```
        subroutine integration_results_read (results, unit)
          type(integration_results_t), intent(out) :: results
          integer, intent(in) :: unit
          character(80) :: buffer
          character :: equals
          integer :: pass, it
          read (unit, *)  buffer
          if (trim (adjustl (buffer)) /= "begin(integration_results)") then
             call read_err ();  return
          end if
          read (unit, *)  buffer, equals, results%n_pass
          read (unit, *)  buffer, equals, results%n_it
          allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
          allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
          it = 0
          do pass = 1, results%n_pass
             read (unit, *)  buffer
             if (trim (adjustl (buffer)) /= "begin(integration_pass)") then
                call read_err ();  return
             end if
             READ_ENTRIES: do
                read (unit, *)  buffer
                if (trim (adjustl (buffer)) /= "begin(iteration)") then
                   exit READ_ENTRIES
                end if
                it = it + 1
                call integration_entry_read (results%entry(it), unit)
                read (unit, *)  buffer
                if (trim (adjustl (buffer)) /= "end(iteration)") then
                   call read_err (); return
```

```
            end if
        end do READ_ENTRIES
        if (trim (adjustl (buffer)) /= "end(integration_pass)") then
            call read_err (); return
        end if
        results%average(pass) = compute_average (results%entry, pass)
    end do
    read (unit, *)  buffer
    if (trim (adjustl (buffer)) /= "end(integration_results)") then
        call read_err (); return
    end if
  contains
    subroutine read_err ()
      call msg_fatal ("Reading integration results from file: syntax error")
    end subroutine read_err
  end subroutine integration_results_read
```

Check integration results for consistency. We compare against an array of pass indices and call numbers. If there is a difference, up to the number of iterations done so far, we return failure. Dummy entries (where **pass** = 0) are ignored.

⟨*Integration results: procedures*⟩+≡
```
    function integration_results_iterations_are_consistent &
         (results, pass, n_calls) result (flag)
    logical :: flag
    type(integration_results_t), intent(in) :: results
    integer, dimension(:), intent(in) :: pass, n_calls
    integer :: n_it
    n_it = results%n_it
    flag = size (pass) >= n_it .and. size (n_calls) >= n_it
    if (flag) then
        flag = all (results%entry(:n_it)%pass == pass(:n_it) &
                    .and. &
                    (results%entry(:n_it)%n_calls == n_calls(:n_it) &
                     .or. &
                     results%entry(:n_it)%process_type == PRC_UNKNOWN))
    end if
    end function integration_results_iterations_are_consistent
```

Discard all results starting from the specified iteration.

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_discard (results, it)
      type(integration_results_t), intent(inout) :: results
      integer, intent(in) :: it
      if (it <= results%n_it) then
        select case (it)
        case (:1)
           results%n_it = 0
           results%n_pass = 0
           results%current_pass = 0
        case default
           results%n_it = it - 1
           results%n_pass = maxval (results%entry(1:results%n_it)%pass)
           results%current_pass = results%n_pass
```

```
          end select
       end if
    end subroutine integration_results_discard
```

Expand the list of entries if the limit has been reached:

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: expand => integration_results_expand
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_expand (results)
      class(integration_results_t), intent(inout) :: results
      type(integration_entry_t), dimension(:), allocatable :: entry_tmp
      if (results%n_it == size (results%entry)) then
         allocate (entry_tmp (results%n_it))
         entry_tmp = results%entry
         deallocate (results%entry)
         allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
         results%entry(:results%n_it) = entry_tmp
         deallocate (entry_tmp)
      end if
      if (results%n_pass == size (results%average)) then
         allocate (entry_tmp (results%n_pass))
         entry_tmp = results%average
         deallocate (results%average)
         allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
         results%average(:results%n_pass) = entry_tmp
         deallocate (entry_tmp)
      end if
    end subroutine integration_results_expand
```

Increment the `current_pass` counter. Can be done before integration; after integration, the recording method may use the value of this counter to define the entry.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: new_pass => integration_results_new_pass
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_new_pass (results)
      class(integration_results_t), intent(inout) :: results
      results%current_pass = results%current_pass + 1
    end subroutine integration_results_new_pass
```

Append a new entry to the list and, if appropriate, compute the average.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: append_entry => integration_results_append_entry
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_append_entry (results, entry)
      class(integration_results_t), intent(inout) :: results
      type(integration_entry_t), intent(in), optional :: entry
      if (results%n_it == 0) then
         results%n_it = 1
         results%n_pass = 1
      else
```

```
        call results%expand ()
        if (present (entry)) then
           if (entry%pass /= results%entry(results%n_it)%pass) &
                 results%n_pass = results%n_pass + 1
        end if
        results%n_it = results%n_it + 1
     end if
     if (present (entry)) then
        results%entry(results%n_it) = entry
        results%average(results%n_pass) = &
              compute_average (results%entry, entry%pass)
     end if
  end subroutine integration_results_append_entry
```

Enter results into the results list. For the error value, we may compare them with a given threshold. This guards against numerical noise, if the exact error would be zero.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: append => integration_results_append
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_append (results, &
        n_it, n_calls, &
        integral, error, efficiency, &
        chain_weights)
      class(integration_results_t), intent(inout) :: results
      integer, intent(in) :: n_it, n_calls
      real(default), intent(in) :: integral, error, efficiency
      real(default), dimension(:), intent(in), optional :: chain_weights
      logical :: improved
      type(integration_entry_t) :: entry
      real(default) :: err_checked
      if (results%n_it /= 0) then
         improved = abs(accuracy (integral, error, n_calls)) &
               < abs(integration_entry_get_accuracy (results%entry(results%n_it)))
      else
         improved = .true.
      end if
      if (abs (error) >= results%error_threshold) then
         err_checked = error
      else
         err_checked = 0
      end if
      call integration_entry_init (entry, &
           results%process_type, results%current_pass, &
           results%n_it+1, n_it, n_calls, improved, &
           integral, err_checked, efficiency, &
           chain_weights=chain_weights)
      call results%append_entry (entry)
    end subroutine integration_results_append
```

Enter an empty result into the results list.

⟨*Integration results: public*⟩+≡

```
    public :: integration_results_append_null
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_append_null (results, pass, n_it)
      type(integration_results_t), intent(inout) :: results
      integer, intent(in) :: pass, n_it
      type(integration_entry_t) :: entry
      call integration_entry_init (entry, &
          PRC_UNKNOWN, results%current_pass, n_it, 1, 0, .false., &
          0._default, 0._default, 0._default)
      call results%append_entry (entry)
    end subroutine integration_results_append_null
```

Record an integration pass executed by an `mci` integrator object.

There is a tolerance below we treat an error (relative to the integral) as zero.

⟨*Limits: public parameters*⟩+≡
```
    real(default), parameter, public :: INTEGRATION_ERROR_TOLERANCE = 1e-10
```

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: record => integration_results_record
```

⟨*Integration results: procedures*⟩+≡
```
    subroutine integration_results_record &
        (object, n_it, n_calls, integral, error, efficiency, &
         chain_weights, suppress)
      class(integration_results_t), intent(inout) :: object
      integer, intent(in) :: n_it, n_calls
      real(default), intent(in) :: integral, error, efficiency
      real(default), dimension(:), intent(in), optional :: chain_weights
      real(default) :: err
      logical, intent(in), optional :: suppress

      if (abs (error) >= abs (integral) * INTEGRATION_ERROR_TOLERANCE) then
         err = error
      else
         err = 0
      end if
      call object%append (n_it, n_calls, integral, err, efficiency, chain_weights)
      call object%display_current (suppress)
    end subroutine integration_results_record
```

15.1.3   Access results

Return true if the results object has entries.

⟨*Integration results: integration results: TBP*⟩+≡
```
    procedure :: exist => integration_results_exist
```

⟨*Integration results: procedures*⟩+≡
```
    function integration_results_exist (results) result (flag)
      logical :: flag
      class(integration_results_t), intent(in) :: results
      flag = results%n_pass > 0
    end function integration_results_exist
```

Retrieve information from the results record. If `last` is set and true, take the last iteration. If `it` is set instead, take this iteration. If `pass` is set, take this average. If none is set, take the final average.

If the result would be invalid, the entry is not assigned. Due to default initialization, this returns a null entry.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: get_entry => results_get_entry
```

⟨*Integration results: procedures*⟩+≡
```
function results_get_entry (results, last, it, pass) result (entry)
  class(integration_results_t), intent(in) :: results
  type(integration_entry_t) :: entry
  logical, intent(in), optional :: last
  integer, intent(in), optional :: it, pass
  if (present (last)) then
     if (allocated (results%entry) .and. results%n_it > 0) then
        entry = results%entry(results%n_it)
     else
        call error ()
     end if
  else if (present (it)) then
     if (allocated (results%entry) .and. it > 0 .and. it <= results%n_it) then
        entry = results%entry(it)
     else
        call error ()
     end if
  else if (present (pass)) then
     if (allocated (results%average) &
          .and. pass > 0 .and. pass <= results%n_pass) then
        entry = results%average (pass)
     else
        call error ()
     end if
  else
     if (allocated (results%average) .and. results%n_pass > 0) then
        entry = results%average (results%n_pass)
     else
        call error ()
     end if
  end if
contains
  subroutine error ()
    call msg_fatal ("Requested integration result is not available")
  end subroutine error
end function results_get_entry
```

The individual procedures. The `results` record should have the `target` attribute, but only locally within the function.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: get_n_calls => integration_results_get_n_calls
procedure :: get_integral => integration_results_get_integral
procedure :: get_error => integration_results_get_error
procedure :: get_accuracy => integration_results_get_accuracy
```

```
      procedure :: get_chi2 => integration_results_get_chi2
      procedure :: get_efficiency => integration_results_get_efficiency
```

⟨*Integration results: procedures*⟩+≡

```
    function integration_results_get_n_calls (results, last, it, pass) &
         result (n_calls)
      class(integration_results_t), intent(in), target :: results
      integer :: n_calls
      logical, intent(in), optional :: last
      integer, intent(in), optional :: it, pass
      n_calls = integration_entry_get_n_calls &
           (results%get_entry (last, it, pass))
    end function integration_results_get_n_calls

    function integration_results_get_integral (results, last, it, pass) &
         result (integral)
      class(integration_results_t), intent(in), target :: results
      real(default) :: integral
      logical, intent(in), optional :: last
      integer, intent(in), optional :: it, pass
      integral = integration_entry_get_integral &
           (results%get_entry (last, it, pass))
    end function integration_results_get_integral

    function integration_results_get_error (results, last, it, pass) &
         result (error)
      class(integration_results_t), intent(in), target :: results
      real(default) :: error
      logical, intent(in), optional :: last
      integer, intent(in), optional :: it, pass
      error = integration_entry_get_error &
           (results%get_entry (last, it, pass))
    end function integration_results_get_error

    function integration_results_get_accuracy (results, last, it, pass) &
         result (accuracy)
      class(integration_results_t), intent(in), target :: results
      real(default) :: accuracy
      logical, intent(in), optional :: last
      integer, intent(in), optional :: it, pass
      accuracy = integration_entry_get_accuracy &
           (results%get_entry (last, it, pass))
    end function integration_results_get_accuracy

    function integration_results_get_chi2 (results, last, it, pass) &
         result (chi2)
      class(integration_results_t), intent(in), target :: results
      real(default) :: chi2
      logical, intent(in), optional :: last
      integer, intent(in), optional :: it, pass
      chi2 = integration_entry_get_chi2 &
           (results%get_entry (last, it, pass))
    end function integration_results_get_chi2

    function integration_results_get_efficiency (results, last, it, pass) &
```

```
      result (efficiency)
    class(integration_results_t), intent(in), target :: results
    real(default) :: efficiency
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    efficiency = integration_entry_get_efficiency &
         (results%get_entry (last, it, pass))
  end function integration_results_get_efficiency
```

Return the last pass index and the index of the last iteration *within* the last pass. The third routine returns the absolute index of the last iteration.

⟨*Integration results: procedures*⟩+≡
```
    function integration_results_get_current_pass (results) result (pass)
      integer :: pass
      type(integration_results_t), intent(in) :: results
      pass = results%n_pass
    end function integration_results_get_current_pass

    function integration_results_get_current_it (results) result (it)
      integer :: it
      type(integration_results_t), intent(in) :: results
      if (allocated (results%entry)) then
         it = count (results%entry(1:results%n_it)%pass == results%n_pass)
      else
         it = 0
      end if
    end function integration_results_get_current_it

    function integration_results_get_last_it (results) result (it)
      integer :: it
      type(integration_results_t), intent(in) :: results
      it = results%n_it
    end function integration_results_get_last_it
```

Return the index of the best iteration (lowest accuracy value) within the current pass. If none qualifies, return zero.

⟨*Integration results: procedures*⟩+≡
```
    function integration_results_get_best_it (results) result (it)
      integer :: it
      type(integration_results_t), intent(in) :: results
      integer :: i
      real(default) :: acc, acc_best
      acc_best = -1
      it = 0
      do i = 1, results%n_it
         if (results%entry(i)%pass == results%n_pass) then
            acc = integration_entry_get_accuracy (results%entry(i))
            if (acc_best < 0 .or. acc <= acc_best) then
               acc_best = acc
               it = i
            end if
         end if
      end do
```

```
            end function integration_results_get_best_it
```

Compute the MD5 sum by printing everything and checksumming the resulting file.

⟨*Integration results: procedures*⟩+≡
```
      function integration_results_get_md5sum (results) result (md5sum_results)
        character(32) :: md5sum_results
        type(integration_results_t), intent(in) :: results
        integer :: u
        u = free_unit ()
        open (unit = u, status = "scratch", action = "readwrite")
        call integration_results_write (results, u, verbose=.true.)
        rewind (u)
        md5sum_results = md5sum (u)
        close (u)
      end function integration_results_get_md5sum
```

This is (ab)used to suppress numerical noise when integrating constant matrix elements.

⟨*Integration results: integration results: TBP*⟩+≡
```
      procedure :: pacify => integration_results_pacify
```

⟨*Integration results: procedures*⟩+≡
```
      subroutine integration_results_pacify (results, efficiency_reset)
        class(integration_results_t), intent(inout) :: results
        logical, intent(in), optional :: efficiency_reset
        integer :: i
        logical :: reset
        reset = .false.
        if (present (efficiency_reset))  reset = efficiency_reset
        if (allocated (results%entry)) then
           do i = 1, size (results%entry)
              call pacify (results%entry(i)%error, &
                   results%entry(i)%integral * 1.E-9_default)
              if (reset)  results%entry(i)%efficiency = 1
           end do
        end if
        if (allocated (results%average)) then
           do i = 1, size (results%average)
              call pacify (results%average(i)%error, &
                   results%average(i)%integral * 1.E-9_default)
              if (reset)  results%average(i)%efficiency = 1
           end do
        end if
      end subroutine integration_results_pacify
```

15.1.4   Results display

Write a driver file for history visualization.

The ratio of y range over y value must not become too small, otherwise we run into an arithmetic overflow in GAMELAN. 2% appears to be safe.

⟨*Limits: public parameters*⟩+≡

```
        real, parameter, public :: GML_MIN_RANGE_RATIO = 0.02
⟨Integration results: public⟩+≡
        public :: integration_results_write_driver

⟨Integration results: procedures⟩+≡
      subroutine integration_results_write_driver (results, filename, eff_reset)
        type(integration_results_t), intent(inout) :: results
        type(string_t), intent(in) :: filename
        logical, intent(in), optional :: eff_reset
        type(string_t) :: file_tex
        integer :: unit
        integer :: n, i, n_pass, pass
        integer, dimension(:), allocatable :: ipass
        real(default) :: ymin, ymax, yavg, ydif, y0, y1
        logical :: reset
        file_tex = filename // ".tex"
        unit = free_unit ()
        open (unit=unit, file=char(file_tex), action="write", status="replace")
        reset = .false.; if (present (eff_reset))  reset = eff_reset
        n = results%n_it
        n_pass = results%n_pass
        allocate (ipass (results%n_pass))
        ipass(1) = 0
        pass = 2
        do i = 1, n-1
           if (integration_entry_get_pass (results%entry(i)) &
                /= integration_entry_get_pass (results%entry(i+1))) then
              ipass(pass) = i
              pass = pass + 1
           end if
        end do
        ymin = minval (integration_entry_get_integral (results%entry(:n)) &
                       - integration_entry_get_error (results%entry(:n)))
        ymax = maxval (integration_entry_get_integral (results%entry(:n)) &
                       + integration_entry_get_error (results%entry(:n)))
        yavg = (ymax + ymin) / 2
        ydif = (ymax - ymin)
        if (ydif * 1.5 > GML_MIN_RANGE_RATIO * yavg) then
           y0 = yavg - ydif * 0.75
           y1 = yavg + ydif * 0.75
        else
           y0 = yavg * (1 - GML_MIN_RANGE_RATIO / 2)
           y1 = yavg * (1 + GML_MIN_RANGE_RATIO / 2)
        end if
        write (unit, "(A)") "\documentclass{article}"
        write (unit, "(A)") "\usepackage{a4wide}"
        write (unit, "(A)") "\usepackage{gamelan}"
        write (unit, "(A)") "\usepackage{amsmath}"
        write (unit, "(A)") ""
        write (unit, "(A)") "\begin{document}"
        write (unit, "(A)") "\begin{gmlfile}"
        write (unit, "(A)") "\section*{Integration Results Display}"
        write (unit, "(A)") ""
        write (unit, "(A)") "Process: \verb|" // char (filename) // "|"
```

```
write (unit, "(A)") ""
write (unit, "(A)") "\vspace*{2\baselineskip}"
write (unit, "(A)") "\unitlength 1mm"
write (unit, "(A)") "\begin{gmlcode}"
write (unit, "(A)") "  picture sym;  sym = fshape (circle scaled 1mm)();"
write (unit, "(A)") "  color col.band;  col.band = 0.9white;"
write (unit, "(A)") "  color col.eband;  col.eband = 0.98white;"
write (unit, "(A)") "\end{gmlcode}"
write (unit, "(A)") "\begin{gmlgraph*}(130,180)[history]"
write (unit, "(A)") "  setup (linear, linear);"
write (unit, "(A,I0,A)") "  history.n_pass = ", n_pass, ";"
write (unit, "(A,I0,A)") "  history.n_it   = ", n, ";"
write (unit, "(A,A,A)") "  history.y0 = #""", char (mp_format (y0)), """;"
write (unit, "(A,A,A)") "  history.y1 = #""", char (mp_format (y1)), """;"
write (unit, "(A)") &
     "  graphrange (#0.5, history.y0), (#(n+0.5), history.y1);"
do pass = 1, n_pass
   write (unit, "(A,I0,A,I0,A)") &
         "  history.pass[", pass, "] = ", ipass(pass), ";"
   write (unit, "(A,I0,A,A,A)") &
         "  history.avg[", pass, "] = #""", &
         char (mp_format &
           (integration_entry_get_integral (results%average(pass)))), &
         """;"
   write (unit, "(A,I0,A,A,A)") &
         "  history.err[", pass, "] = #""", &
         char (mp_format &
           (integration_entry_get_error (results%average(pass)))), &
         """;"
   write (unit, "(A,I0,A,A,A)") &
         "  history.chi[", pass, "] = #""", &
         char (mp_format &
           (integration_entry_get_chi2 (results%average(pass)))), &
         """;"
end do
write (unit, "(A,I0,A,I0,A)") &
     "  history.pass[", n_pass + 1, "] = ", n, ";"
write (unit, "(A)") "  for i = 1 upto history.n_pass:"
write (unit, "(A)") "    if history.chi[i] greater one:"
write (unit, "(A)") "    fill plot ("
write (unit, "(A)") &
     "        (#(history.pass[i]  +.5), " &
   // "history.avg[i] minus history.err[i] times history.chi[i]),"
write (unit, "(A)") &
     "        (#(history.pass[i+1]+.5), " &
   // "history.avg[i] minus history.err[i] times history.chi[i]),"
write (unit, "(A)") &
     "        (#(history.pass[i+1]+.5), " &
   // "history.avg[i] plus history.err[i] times history.chi[i]),"
write (unit, "(A)") &
     "        (#(history.pass[i]  +.5), " &
   // "history.avg[i] plus history.err[i] times history.chi[i])"
write (unit, "(A)") "    ) withcolor col.eband fi;"
write (unit, "(A)") "    fill plot ("
```

1933

```fortran
    write (unit, "(A)")  &
         "        (#(history.pass[i]  +.5), history.avg[i] minus history.err[i]),"
    write (unit, "(A)")  &
         "        (#(history.pass[i+1]+.5), history.avg[i] minus history.err[i]),"
    write (unit, "(A)")  &
         "        (#(history.pass[i+1]+.5), history.avg[i] plus history.err[i]),"
    write (unit, "(A)")  &
         "        (#(history.pass[i]  +.5), history.avg[i] plus history.err[i])"
    write (unit, "(A)")  "      ) withcolor col.band;"
    write (unit, "(A)")  "    draw plot ("
    write (unit, "(A)")  &
         "        (#(history.pass[i]  +.5), history.avg[i]),"
    write (unit, "(A)")  &
         "        (#(history.pass[i+1]+.5), history.avg[i])"
    write (unit, "(A)")  "       ) dashed evenly;"
    write (unit, "(A)")  "  endfor"
    write (unit, "(A)")  "  for i = 1 upto history.n_pass + 1:"
    write (unit, "(A)")  "    draw plot ("
    write (unit, "(A)")  &
         "        (#(history.pass[i]+.5), history.y0),"
    write (unit, "(A)")  &
         "        (#(history.pass[i]+.5), history.y1)"
    write (unit, "(A)")  "       ) dashed withdots;"
    write (unit, "(A)")  "  endfor"
    do i = 1, n
       write (unit, "(A,I0,A,A,A,A,A)")  "  plot (history) (#", &
          i, ", #""", &
          char (mp_format (integration_entry_get_integral (results%entry(i)))),&
          """) vbar #""", &
          char (mp_format (integration_entry_get_error (results%entry(i)))), &
          """;"
    end do
    write (unit, "(A)")  "  draw piecewise from (history) " &
      // "withsymbol sym;"
    write (unit, "(A)")  "  fullgrid.lr (5,20);"
    write (unit, "(A)")  "  standardgrid.bt (n);"
    write (unit, "(A)")  "  begingmleps ""Whizard-Logo.eps"";"
    write (unit, "(A)")  "    base := (120*unitlength,170*unitlength);"
    write (unit, "(A)")  "    height := 9.6*unitlength;"
    write (unit, "(A)")  "    width := 11.2*unitlength;"
    write (unit, "(A)")  "  endgmleps;"
    write (unit, "(A)")  "\end{gmlgraph*}"
    write (unit, "(A)")  "\end{gmlfile}"
    write (unit, "(A)")  "\clearpage"
    write (unit, "(A)")  "\begin{verbatim}"
    if (reset) then
       call results%pacify (reset)
    end if
    call integration_results_write (results, unit)
    write (unit, "(A)")  "\end{verbatim}"
    write (unit, "(A)")  "\end{document}"
    close (unit)
  end subroutine integration_results_write_driver
```

Call L^AT_EX and Metapost for the history driver file, and convert to PS and PDF.

⟨*Integration results: public*⟩+≡

```
    public :: integration_results_compile_driver
```

⟨*Integration results: procedures*⟩+≡

```
    subroutine integration_results_compile_driver (results, filename, os_data)
      type(integration_results_t), intent(in) :: results
      type(string_t), intent(in) :: filename
      type(os_data_t), intent(in) :: os_data
      integer :: unit_dev, status
      type(string_t) :: file_tex, file_dvi, file_ps, file_pdf, file_mp
      type(string_t) :: setenv_tex, setenv_mp, pipe, pipe_dvi
      if (.not. os_data%event_analysis) then
         call msg_warning ("Skipping integration history display " &
             // "because latex or mpost is not available")
         return
      end if
      file_tex = filename // ".tex"
      file_dvi = filename // ".dvi"
      file_ps = filename // ".ps"
      file_pdf = filename // ".pdf"
      file_mp = filename // ".mp"
      call msg_message ("Creating integration history display "&
          // char (file_ps) // " and " // char (file_pdf))
      BLOCK: do
         unit_dev = free_unit ()
         open (file = "/dev/null", unit = unit_dev, &
               action = "write", iostat = status)
         if (status /= 0) then
           pipe = ""
           pipe_dvi = ""
         else
           pipe = " > /dev/null"
           pipe_dvi = " 2>/dev/null 1>/dev/null"
         end if
         close (unit_dev)
         if (os_data%whizard_texpath /= "") then
            setenv_tex = &
                "TEXINPUTS=" // os_data%whizard_texpath // ":$TEXINPUTS "
            setenv_mp = &
                "MPINPUTS=" // os_data%whizard_texpath // ":$MPINPUTS "
         else
            setenv_tex = ""
            setenv_mp = ""
         end if
         call os_system_call (setenv_tex // os_data%latex // " " // &
             file_tex // pipe, status)
         if (status /= 0)  exit BLOCK
         if (os_data%gml /= "") then
            call os_system_call (setenv_mp // os_data%gml // " " // &
                file_mp // pipe, status)
         else
            call msg_error ("Could not use GAMELAN/MetaPOST.")
            exit BLOCK
         end if
```

```
         if (status /= 0)  exit BLOCK
         call os_system_call (setenv_tex // os_data%latex // " " // &
              file_tex // pipe, status)
         if (status /= 0)  exit BLOCK
         if (os_data%event_analysis_ps) then
            call os_system_call (os_data%dvips // " " // &
               file_dvi // pipe_dvi, status)
            if (status /= 0)  exit BLOCK
         else
            call msg_warning ("Skipping PostScript generation because dvips " &
                 // "is not available")
            exit BLOCK
         end if
         if (os_data%event_analysis_pdf) then
            call os_system_call (os_data%ps2pdf // " " // &
                 file_ps, status)
            if (status /= 0)  exit BLOCK
         else
            call msg_warning ("Skipping PDF generation because ps2pdf " &
                 // "is not available")
            exit BLOCK
         end if
         exit BLOCK
      end do BLOCK
      if (status /= 0) then
         call msg_error ("Unable to compile integration history display")
      end if
   end subroutine integration_results_compile_driver
```

15.2   Abstract process core

In this module we provide abstract data types for process classes. Each process
class represents a set of processes which are handled by a common "method",
e.g., by the O'MEGA matrix-element generator. The process class is also able to
select a particular implementation for the phase-space and integration modules.

For a complete implementation of a process class, we have to provide exten-
sions of the following abstract types:

prc_core_def_t process and matrix-element configuration

prc_writer_t (optional) writing external matrix-element code

prc_driver_t accessing the matrix element (internal or external)

prc_core_t evaluating kinematics and matrix element. The process core also
   selects phase-space and integrator implementations as appropriate for the
   process class and configuration.

In the actual process-handling data structures, each process component con-
tains an instance of such a process class as its core. This allows us to keep the
processes module below, which supervises matrix-element evaluation, integra-
tion, and event generation, free of any reference to concrete implementations
(for the process class, phase space, and integrator).

There are no unit tests, these are deferred to the `processes` module.

⟨`prc_core.f90`⟩≡
  ⟨*File header*⟩
  `module prc_core`

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    `use diagnostics !NODEP!`
    `use lorentz !NODEP!`
    `use interactions`

    `use sf_base`
    `use process_constants`
    `use mci_base`

    `use prc_core_def`
    `use process_libraries`

  ⟨*Standard module head*⟩

  ⟨*Prc core: public*⟩

  ⟨*Prc core: types*⟩

  ⟨*Prc core: interfaces*⟩

  `contains`

  ⟨*Prc core: procedures*⟩

  `end module prc_core`

15.2.1 The process core

The process core is of abstract data type. Different types of matrix element will be represented by different implementations.

⟨*Prc core: public*⟩≡
    `public :: prc_core_t`

⟨*Prc core: types*⟩≡
    `type, abstract :: prc_core_t`
      `class(prc_core_def_t), pointer :: def => null ()`
      `logical :: data_known = .false.`
      `type(process_constants_t) :: data`
      `class(prc_core_driver_t), allocatable :: driver`
      `logical :: use_color_factors = .false.`
      `integer :: nc = 3`
     `contains`
     ⟨*Prc core: process core: TBP*⟩
    `end type prc_core_t`

In any case there must be an output routine.

⟨*Prc core: process core: TBP*⟩≡
```
    procedure(prc_core_write), deferred :: write
```

⟨*Prc core: interfaces*⟩≡
```
    abstract interface
       subroutine prc_core_write (object, unit)
         import
         class(prc_core_t), intent(in) :: object
         integer, intent(in), optional :: unit
       end subroutine prc_core_write
    end interface
```

For initialization, we assign a pointer to the process entry in the relevant library. This allows us to access all process functions via the implementation of `prc_core_t`.

We declare the `object` as `intent(inout)`, since just after allocation it may be useful to store some extra data in the object, which we can then use in the actual initialization. This applies to extensions of `prc_core` which override the `init` method.

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure :: init => prc_core_init
    procedure :: base_init => prc_core_init
```

⟨*Prc core: procedures*⟩≡
```
    subroutine prc_core_init (object, def, lib, id, i_component)
      class(prc_core_t), intent(inout) :: object
      class(prc_core_def_t), intent(in), target :: def
      type(process_library_t), intent(in), target :: lib
      type(string_t), intent(in) :: id
      integer, intent(in) :: i_component
      object%def => def
      call lib%connect_process (id, i_component, object%data, object%driver)
      object%data_known = .true.
    end subroutine prc_core_init
```

Return true if the matrix element generation was successful. This can be tested by looking at the number of generated flavor states, which should be nonzero.

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure :: has_matrix_element => prc_core_has_matrix_element
```

⟨*Prc core: procedures*⟩+≡
```
    function prc_core_has_matrix_element (object) result (flag)
      class(prc_core_t), intent(in) :: object
      logical :: flag
      flag = object%data%n_flv /= 0
    end function prc_core_has_matrix_element
```

Return true if a MC dataset should be attached to this process component. False if it shares the dataset with another component.

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_get_flag), deferred :: needs_mcset
```

1938

⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       function prc_core_get_flag (object) result (flag)
         import
         class(prc_core_t), intent(in) :: object
         logical :: flag
       end function prc_core_get_flag
    end interface
```

Return an integer number
⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       function prc_core_get_integer (object) result (i)
         import
         class(prc_core_t), intent(in) :: object
         integer :: i
       end function prc_core_get_integer
    end interface
```

Return the number of distinct terms requested by this process component.
⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_get_integer), deferred :: get_n_terms
```
Tell whether a particular combination of flavor/helicity/color state is allowed for the matrix element.
⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_is_allowed), deferred :: is_allowed
```

⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       function prc_core_is_allowed (object, i_term, f, h, c) result (flag)
         import
         class(prc_core_t), intent(in) :: object
         integer, intent(in) :: i_term, f, h, c
         logical :: flag
       end function prc_core_is_allowed
    end interface
```

Set the constant process data for a specific term. By default, these are the constants stored inside the object, ignoring the term index. Type extensions may override this and provide term-specific data.
⟨*Prc core: process core: TBP*⟩+≡
```
    procedure :: get_constants => prc_core_get_constants
```

⟨*Prc core: procedures*⟩+≡
```
    subroutine prc_core_get_constants (object, data, i_term)
      class(prc_core_t), intent(in) :: object
      type(process_constants_t), intent(out) :: data
      integer, intent(in) :: i_term
      data = object%data
    end subroutine prc_core_get_constants
```

The strong coupling is not among the process constants. The default implementation is to return a negative number, which indicates that α_s is not available. This may be overridden by an implementation that provides an (event-specific) value. The value can be stored in the `tmp` workspace.

⟨*Prc core: process core: TBP*⟩+≡
```
procedure :: get_alpha_s => prc_core_get_alpha_s
```

⟨*Prc core: procedures*⟩+≡
```
function prc_core_get_alpha_s (object, tmp) result (alpha)
  class(prc_core_t), intent(in) :: object
  class(workspace_t), intent(in), allocatable :: tmp
  real(default) :: alpha
  alpha = -1
end function prc_core_get_alpha_s
```

Allocate the workspace associated to a process component. The default is that there is no workspace, so we do nothing. A type extension may override this and allocate a workspace object of appropriate type, which can be used in further calculations.

In any case, the `intent(out)` attribute deletes any previously allocated workspace.

⟨*Prc core: process core: TBP*⟩+≡
```
procedure :: allocate_workspace => prc_core_ignore_workspace
```

⟨*Prc core: procedures*⟩+≡
```
subroutine prc_core_ignore_workspace (object, tmp)
  class(prc_core_t), intent(in) :: object
  class(workspace_t), intent(inout), allocatable :: tmp
end subroutine prc_core_ignore_workspace
```

Initialize the structure-function instance that corresponds to a process component. In ordinary cases, this amounts to a straightforward copy of the given template, but the process core may also choose to modify the setup, and it may access its own workspace.

The `sf_chain_instance` is the object that we want to prepare. The `sf_chain` argument is a template for its structure. For the initialization, we also need the number of channels `n_channel`.

Note: crash with nagfor 5.3.1 if `sf_chain_instance` is declared `intent(out)`, as would be more appropriate.

⟨*Prc core: process core: TBP*⟩+≡
```
procedure :: init_sf_chain => prc_core_init_sf_chain
```

⟨*Prc core: procedures*⟩+≡
```
subroutine prc_core_init_sf_chain &
    (object, sf_chain_instance, sf_chain, n_channel, tmp)
  class(prc_core_t), intent(in) :: object
  type(sf_chain_instance_t), intent(inout), target :: sf_chain_instance
  type(sf_chain_t), intent(in), target :: sf_chain
  integer, intent(in) :: n_channel
  class(workspace_t), intent(inout), allocatable :: tmp
  call sf_chain_instance%init (sf_chain, n_channel)
end subroutine prc_core_init_sf_chain
```

Compute the momenta in the hard interaction, taking the seed kinematics as input. The `i_term` index tells us which term we want to compute. (The standard method is to just transfer the momenta to the hard interaction.)

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_compute_hard_kinematics), deferred :: &
        compute_hard_kinematics
```

⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_core_compute_hard_kinematics &
            (object, p_seed, i_term, int_hard, tmp)
         import
         class(prc_core_t), intent(in) :: object
         type(vector4_t), dimension(:), intent(in) :: p_seed
         integer, intent(in) :: i_term
         type(interaction_t), intent(inout) :: int_hard
         class(workspace_t), intent(inout), allocatable :: tmp
       end subroutine prc_core_compute_hard_kinematics
    end interface
```

Compute the momenta in the effective interaction, taking the hard kinematics as input. (This is called only if parton recombination is to be applied for the process variant.)

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_compute_eff_kinematics), deferred :: &
        compute_eff_kinematics
```

⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_core_compute_eff_kinematics &
            (object, i_term, int_hard, int_eff, tmp)
         import
         class(prc_core_t), intent(in) :: object
         integer, intent(in) :: i_term
         type(interaction_t), intent(in) :: int_hard
         type(interaction_t), intent(inout) :: int_eff
         class(workspace_t), intent(inout), allocatable :: tmp
       end subroutine prc_core_compute_eff_kinematics
    end interface
```

Recover the missing pieces. We know the incoming momenta of the `p_seed` array and the outgoing momenta of the `int_eff` interaction. We have to recover the outgoing momenta of `p_seed` and the incoming momenta of `int_eff`.

(The trivial case is that these are identical.)

Furthermore, if `int_hard` and `int_eff` are not aliased, we have to set the momenta there. In the trivial case, `int_eff` is a pointer to `int_hard`, so `int_hard` should not be touched at all.

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_recover_kinematics), deferred :: &
        recover_kinematics
```

⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       subroutine prc_core_recover_kinematics &
            (object, p_seed, int_hard, int_eff, tmp)
          import
          class(prc_core_t), intent(in) :: object
          type(vector4_t), dimension(:), intent(inout) :: p_seed
          type(interaction_t), intent(inout) :: int_hard
          type(interaction_t), intent(inout) :: int_eff
          class(workspace_t), intent(inout), allocatable :: tmp
       end subroutine prc_core_recover_kinematics
    end interface
```

The process core must implement this function. Here, `j` is the index of the particular term we want to compute. The amplitude may depend on the factorization and renormalization scales.

The `tmp` (workspace) argument may be used if it is provided by the caller. Otherwise, the routine should compute the result directly.

⟨*Prc core: process core: TBP*⟩+≡
```
    procedure(prc_core_compute_amplitude), deferred :: compute_amplitude
```

⟨*Prc core: interfaces*⟩+≡
```
    abstract interface
       function prc_core_compute_amplitude &
            (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
          import
          class(prc_core_t), intent(in) :: object
          integer, intent(in) :: j
          type(vector4_t), dimension(:), intent(in) :: p
          integer, intent(in) :: f, h, c
          real(default), intent(in) :: fac_scale, ren_scale
          class(workspace_t), intent(inout), allocatable, optional :: tmp
          complex(default) :: amp
       end function prc_core_compute_amplitude
    end interface
```

15.2.2 Storage for intermediate results

The abstract `workspace_t` type allows process cores to set up temporary workspace. The object is an extra argument for each of the individual calculations between kinematics setup and matrix-element evaluation.

⟨*Prc core: public*⟩+≡
```
    public :: workspace_t
```

⟨*Prc core: types*⟩+≡
```
    type, abstract :: workspace_t
     contains
       procedure(workspace_write), deferred :: write
    end type workspace_t
```

For debugging, we should at least have an output routine.

⟨*Prc core: interfaces*⟩+≡
```
abstract interface
   subroutine workspace_write (object, unit)
     import
     class(workspace_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine workspace_write
end interface
```

15.2.3   Helicity selection data

This is intended for use with `O'MEGA`, but may also be made available to other process methods. We set thresholds for counting the times a specific helicity amplitude is zero. When the threshold is reached, we skip this amplitude in subsequent calls.

For initializing the helicity counters, we need an object that holds the two parameters, the threshold (large real number) and the cutoff (integer).

A helicity value suppressed by more than `threshold` (a value which multiplies `epsilon`, to be compared with the average of the current amplitude, default is 10^{10}) is treated as zero. A matrix element is assumed to be zero and not called again if it has been zero `cutoff` times.

⟨*Prc core: public*⟩+≡
```
public :: helicity_selection_t
```

⟨*Prc core: types*⟩+≡
```
type :: helicity_selection_t
   logical :: active = .false.
   real(default) :: threshold = 0
   integer :: cutoff = 0
 contains
 ⟨Prc core: helicity selection: TBP⟩
end type helicity_selection_t
```

Output. If the selection is inactive, print nothing.

⟨*Prc core: helicity selection: TBP*⟩≡
```
procedure :: write => helicity_selection_write
```

⟨*Prc core: procedures*⟩+≡
```
subroutine helicity_selection_write (object, unit)
  class(helicity_selection_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  if (object%active) then
     write (u, "(3x,A)")  "Helicity selection data:"
     write (u, "(5x,A,ES17.10)") &
           "threshold =", object%threshold
     write (u, "(5x,A,I0)") &
           "cutoff    = ", object%cutoff
  end if
end subroutine helicity_selection_write
```

15.3 Process observables

We define an abstract `subevt_expr_t` object as an extension of the `subevt_t` type. The object contains a local variable list, variable instances (as targets for pointers in the variable list), and evaluation trees. The evaluation trees reference both the variables and the `subevt`.

There are two instances of the abstract type: one for process instances, one for physical events. Both have a common logical expression `selection` which determines whether the object passes user-defined cuts.

The intention is that we fill the `subevt_t` base object and compute the variables once we have evaluated a kinematical phase space point (or a complete event). We then evaluate the expressions and can use the results in further calculations.

The `process_expr_t` extension contains furthermore scale and weight expressions. The `event_expr_t` extension contains a reweighting-factor expression and a logical expression for event analysis. In practice, we will link the variable list of the `event_obs` object to the variable list of the currently active `process_obs` object, such that the process variables are available to both objects. Event variables are meaningful only for physical events.

⟨`subevt_expr.f90`⟩≡
```
  ⟨File header⟩
  module subevt_expr

  ⟨Use kinds⟩
  ⟨Use strings⟩
  ⟨Use file utils⟩
    use limits, only: FMT_12 !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use os_interface

    use ifiles
    use lexers
    use parser

    use lorentz !NODEP!
    use subevents
    use variables
    use expressions
    use models
    use flavors
    use interactions
    use particles

  ⟨Standard module head⟩

  ⟨Subevt expr: public⟩

  ⟨Subevt expr: types⟩
```

```
    contains
```

⟨*Subevt expr: procedures*⟩

⟨*Subevt expr: tests*⟩

```
    end module subevt_expr
```

15.3.1   Abstract base type

⟨*Subevt expr: types*⟩≡
```
    type, extends (subevt_t), abstract :: subevt_expr_t
      logical :: subevt_filled = .false.
      type(var_list_t) :: var_list
      real(default) :: sqrts_hat = 0
      integer :: n_in = 0
      integer :: n_out = 0
      integer :: n_tot = 0
      logical :: has_selection = .false.
      type(eval_tree_t) :: selection
     contains
     ⟨Subevt expr: subevt expr: TBP⟩
    end type subevt_expr_t
```

Output: Base and extended version. We already have a `write` routine for the
`subevt_t` parent type.

⟨*Subevt expr: subevt expr: TBP*⟩≡
```
    procedure :: base_write => subevt_expr_write
```

⟨*Subevt expr: procedures*⟩≡
```
    subroutine subevt_expr_write (object, unit)
      class(subevt_expr_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "Local variables:"
      call write_separator (u)
      call var_list_write (object%var_list, u, follow_link=.false.)
      call write_separator (u)
      if (object%subevt_filled) then
         call object%subevt_t%write (u)
         if (object%has_selection) then
            call write_separator (u)
            write (u, "(1x,A)")  "Selection expression:"
            call write_separator (u)
            call eval_tree_write (object%selection, u)
         end if
      else
         write (u, "(1x,A)")  "subevt: [undefined]"
      end if
    end subroutine subevt_expr_write
```

Finalizer.

⟨*Subevt expr: subevt expr: TBP*⟩+≡
```
    procedure (subevt_expr_final), deferred :: final
    procedure :: base_final => subevt_expr_final
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine subevt_expr_final (object)
      class(subevt_expr_t), intent(inout) :: object
      call var_list_final (object%var_list)
      if (object%has_selection) then
          call eval_tree_final (object%selection)
      end if
    end subroutine subevt_expr_final
```

15.3.2   Initialization

Initialization: define local variables and establish pointers.

The common variables are `sqrts` (the nominal beam energy, fixed), `sqrts_hat` (the actual energy), `n_in`, `n_out`, and `n_tot` for the `subevt`. With the exception of `sqrts`, all are implemented as pointers to subobjects.

⟨*Subevt expr: subevt expr: TBP*⟩+≡
```
    procedure (subevt_expr_setup_vars), deferred :: setup_vars
    procedure :: base_setup_vars => subevt_expr_setup_vars
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine subevt_expr_setup_vars (expr, sqrts)
      class(subevt_expr_t), intent(inout), target :: expr
      real(default), intent(in) :: sqrts
      call var_list_final (expr%var_list)
      call var_list_append_real (expr%var_list, &
          var_str ("sqrts"), sqrts, &
          locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_real_ptr (expr%var_list, &
          var_str ("sqrts_hat"), expr%sqrts_hat, &
          is_known = expr%subevt_filled, &
          locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_int_ptr (expr%var_list, &
          var_str ("n_in"), expr%n_in, &
          is_known = expr%subevt_filled, &
          locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_int_ptr (expr%var_list, &
          var_str ("n_out"), expr%n_out, &
          is_known = expr%subevt_filled, &
          locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_int_ptr (expr%var_list, &
          var_str ("n_tot"), expr%n_tot, &
          is_known = expr%subevt_filled, &
          locked = .true., verbose = .false., intrinsic = .true.)
    end subroutine subevt_expr_setup_vars
```

Link a variable list to the local one. This could be done event by event, but before evaluating expressions.

⟨Subevt expr: subevt expr: TBP⟩+≡
```
    procedure :: link_var_list => subevt_expr_link_var_list
```

⟨Subevt expr: procedures⟩+≡
```
    subroutine subevt_expr_link_var_list (expr, var_list)
      class(subevt_expr_t), intent(inout) :: expr
      type(var_list_t), intent(in), target :: var_list
      call var_list_link (expr%var_list, var_list)
    end subroutine subevt_expr_link_var_list
```

Compile the selection expression. If the pointer is disassociated, there is no expression.

⟨Subevt expr: subevt expr: TBP⟩+≡
```
    procedure :: setup_selection => subevt_expr_setup_selection
```

⟨Subevt expr: procedures⟩+≡
```
    subroutine subevt_expr_setup_selection (expr, pn_selection)
      class(subevt_expr_t), intent(inout), target :: expr
      type(parse_node_t), intent(in), pointer :: pn_selection
      if (associated (pn_selection)) then
         call eval_tree_init_lexpr (expr%selection, &
              pn_selection, expr%var_list, expr%subevt_t)
         expr%has_selection = .true.
      end if
    end subroutine subevt_expr_setup_selection
```

15.3.3 Evaluation

Reset to initial state, i.e., mark the `subevt` as invalid.

⟨Subevt expr: subevt expr: TBP⟩+≡
```
    procedure :: reset => subevt_expr_reset
    procedure :: base_reset => subevt_expr_reset
```

⟨Subevt expr: procedures⟩+≡
```
    subroutine subevt_expr_reset (expr)
      class(subevt_expr_t), intent(inout) :: expr
      expr%subevt_filled = .false.
    end subroutine subevt_expr_reset
```

Evaluate the selection expression and return the result. There is also a deferred version: this should evaluate the remaining expressions if the event has passed.

⟨Subevt expr: subevt expr: TBP⟩+≡
```
    procedure :: base_evaluate => subevt_expr_evaluate
```

⟨Subevt expr: procedures⟩+≡
```
    subroutine subevt_expr_evaluate (expr, passed)
      class(subevt_expr_t), intent(inout) :: expr
      logical, intent(out) :: passed
      if (expr%has_selection) then
         call eval_tree_evaluate (expr%selection)
```

```
         if (eval_tree_result_is_known (expr%selection)) then
            passed = eval_tree_get_log (expr%selection)
         else
            call msg_error ("Evaluate selection expression: result undefined")
            passed = .false.
         end if
      else
         passed = .true.
      end if
   end subroutine subevt_expr_evaluate
```

15.3.4 Implementation for partonic events

This implementation contains the expressions that we can evaluate for the partonic process during integration.

⟨*Subevt expr: public*⟩≡
```
    public :: parton_expr_t
```

⟨*Subevt expr: types*⟩+≡
```
    type, extends (subevt_expr_t) :: parton_expr_t
       integer, dimension(:), allocatable :: i_beam
       integer, dimension(:), allocatable :: i_in
       integer, dimension(:), allocatable :: i_out
       logical :: has_scale = .false.
       logical :: has_fac_scale = .false.
       logical :: has_ren_scale = .false.
       logical :: has_weight = .false.
       type(eval_tree_t) :: scale
       type(eval_tree_t) :: fac_scale
       type(eval_tree_t) :: ren_scale
       type(eval_tree_t) :: weight
     contains
     ⟨Subevt expr: parton expr: TBP⟩
    end type parton_expr_t
```

Finalizer.

⟨*Subevt expr: parton expr: TBP*⟩≡
```
    procedure :: final => parton_expr_final
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_final (object)
      class(parton_expr_t), intent(inout) :: object
      call object%base_final ()
      if (object%has_scale) then
         call eval_tree_final (object%scale)
      end if
      if (object%has_fac_scale) then
         call eval_tree_final (object%fac_scale)
      end if
      if (object%has_ren_scale) then
         call eval_tree_final (object%ren_scale)
      end if
      if (object%has_weight) then
```

```
        call eval_tree_final (object%weight)
      end if
    end subroutine parton_expr_final
```

Output: continue writing the active expressions, after the common selection expression.

Note: the `prefix` argument is declared in the `write` method of the `subevt_t` base type. Here, it is unused.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: write => parton_expr_write
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_write (object, unit, prefix)
      class(parton_expr_t), intent(in) :: object
      integer, intent(in), optional :: unit
      character(*), intent(in), optional :: prefix
      integer :: u
      u = output_unit (unit)
      call object%base_write (u)
      if (object%subevt_filled) then
         if (object%has_scale) then
            call write_separator (u)
            write (u, "(1x,A)")  "Scale expression:"
            call write_separator (u)
            call eval_tree_write (object%scale, u)
         end if
         if (object%has_fac_scale) then
            call write_separator (u)
            write (u, "(1x,A)")  "Factorization scale expression:"
            call write_separator (u)
            call eval_tree_write (object%fac_scale, u)
         end if
         if (object%has_ren_scale) then
            call write_separator (u)
            write (u, "(1x,A)")  "Renormalization scale expression:"
            call write_separator (u)
            call eval_tree_write (object%ren_scale, u)
         end if
         if (object%has_weight) then
            call write_separator (u)
            write (u, "(1x,A)")  "Weight expression:"
            call write_separator (u)
            call eval_tree_write (object%weight, u)
         end if
      end if
    end subroutine parton_expr_write
```

Define variables.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: setup_vars => parton_expr_setup_vars
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_setup_vars (expr, sqrts)
      class(parton_expr_t), intent(inout), target :: expr
```

```
      real(default), intent(in) :: sqrts
      call expr%base_setup_vars (sqrts)
    end subroutine parton_expr_setup_vars
```

Compile the scale expressions. If a pointer is disassociated, there is no expression.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: setup_scales => parton_expr_setup_scales
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_setup_scales &
        (expr, pn_scale, pn_fac_scale, pn_ren_scale)
      class(parton_expr_t), intent(inout), target :: expr
      type(parse_node_t), intent(in), pointer :: pn_scale
      type(parse_node_t), intent(in), pointer :: pn_fac_scale, pn_ren_scale
      if (associated (pn_scale)) then
         call eval_tree_init_expr (expr%scale, &
              pn_scale, expr%var_list, expr%subevt_t)
         expr%has_scale = .true.
      end if
      if (associated (pn_fac_scale)) then
         call eval_tree_init_expr (expr%fac_scale, &
              pn_fac_scale, expr%var_list, expr%subevt_t)
         expr%has_fac_scale = .true.
      end if
      if (associated (pn_ren_scale)) then
         call eval_tree_init_expr (expr%ren_scale, &
              pn_ren_scale, expr%var_list, expr%subevt_t)
         expr%has_ren_scale = .true.
      end if
    end subroutine parton_expr_setup_scales
```

Compile the weight expression.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: setup_weight => parton_expr_setup_weight
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_setup_weight (expr, pn_weight)
      class(parton_expr_t), intent(inout), target :: expr
      type(parse_node_t), intent(in), pointer :: pn_weight
      if (associated (pn_weight)) then
         call eval_tree_init_expr (expr%weight, &
              pn_weight, expr%var_list, expr%subevt_t)
         expr%has_weight = .true.
      end if
    end subroutine parton_expr_setup_weight
```

Filling the partonic state consists of two parts. The first routine prepares the subevt without assigning momenta. It takes the particles from an `interaction_t`. It needs the indices and flavors for the beam, incoming, and outgoing particles.

We can assume that the particle content of the subevt does not change.

1950

Therefore, we set the event variables `n_in`, `n_out`, `n_tot` already in this initialization step.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: setup_subevt => parton_expr_setup_subevt
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_setup_subevt (expr, int, &
         i_beam, i_in, i_out, f_beam, f_in, f_out)
      class(parton_expr_t), intent(inout) :: expr
      type(interaction_t), intent(in), target :: int
      integer, dimension(:), intent(in) :: i_beam, i_in, i_out
      type(flavor_t), dimension(:), intent(in) :: f_beam, f_in, f_out
      allocate (expr%i_beam (size (i_beam)))
      allocate (expr%i_in (size (i_in)))
      allocate (expr%i_out (size (i_out)))
      expr%i_beam = i_beam
      expr%i_in = i_in
      expr%i_out = i_out
      call interaction_to_subevt (int, &
           expr%i_beam, expr%i_in, expr%i_out, expr%subevt_t)
      call subevt_set_pdg_beam     (expr%subevt_t, flavor_get_pdg (f_beam))
      call subevt_set_pdg_incoming (expr%subevt_t, flavor_get_pdg (f_in))
      call subevt_set_pdg_outgoing (expr%subevt_t, flavor_get_pdg (f_out))
      call subevt_set_p2_beam     (expr%subevt_t, flavor_get_mass (f_beam) ** 2)
      call subevt_set_p2_incoming (expr%subevt_t, flavor_get_mass (f_in)   ** 2)
      call subevt_set_p2_outgoing (expr%subevt_t, flavor_get_mass (f_out)  ** 2)
      expr%n_in  = size (i_in)
      expr%n_out = size (i_out)
      expr%n_tot = expr%n_in + expr%n_out
    end subroutine parton_expr_setup_subevt
```

The second part takes the momenta from the interaction object and thus completes the subevt. The partonic energy can then be computed.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: fill_subevt => parton_expr_fill_subevt
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_fill_subevt (expr, int)
      class(parton_expr_t), intent(inout) :: expr
      type(interaction_t), intent(in), target :: int
      call interaction_momenta_to_subevt (int, &
           expr%i_beam, expr%i_in, expr%i_out, expr%subevt_t)
      expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
      expr%subevt_filled = .true.
    end subroutine parton_expr_fill_subevt
```

Evaluate, if the event passes the selection. For absent expressions we take default values.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: evaluate => parton_expr_evaluate
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_evaluate &
         (expr, passed, scale, fac_scale, ren_scale, weight)
```

```fortran
class(parton_expr_t), intent(inout) :: expr
logical, intent(out) :: passed
real(default), intent(out) :: scale
real(default), intent(out) :: fac_scale
real(default), intent(out) :: ren_scale
real(default), intent(out) :: weight
call expr%base_evaluate (passed)
if (passed) then
   if (expr%has_scale) then
      call eval_tree_evaluate (expr%scale)
      if (eval_tree_result_is_known (expr%scale)) then
         scale = eval_tree_get_real (expr%scale)
      else
         call msg_error ("Evaluate scale expression: result undefined")
         scale = 0
      end if
   else
      scale = expr%sqrts_hat
   end if
   if (expr%has_fac_scale) then
      call eval_tree_evaluate (expr%fac_scale)
      if (eval_tree_result_is_known (expr%fac_scale)) then
         fac_scale = eval_tree_get_real (expr%fac_scale)
      else
         call msg_error ("Evaluate factorization scale expression: &
              &result undefined")
         fac_scale = 0
      end if
   else
      fac_scale = scale
   end if
   if (expr%has_ren_scale) then
      call eval_tree_evaluate (expr%ren_scale)
      if (eval_tree_result_is_known (expr%ren_scale)) then
         ren_scale = eval_tree_get_real (expr%ren_scale)
      else
         call msg_error ("Evaluate renormalization scale expression: &
              &result undefined")
         ren_scale = 0
      end if
   else
      ren_scale = scale
   end if
   if (expr%has_weight) then
      call eval_tree_evaluate (expr%weight)
      if (eval_tree_result_is_known (expr%weight)) then
         weight = eval_tree_get_real (expr%weight)
      else
         call msg_error ("Evaluate weight expression: result undefined")
         weight = 0
      end if
   else
      weight = 1
   end if
```

```
      end if
    end subroutine parton_expr_evaluate
```

Return the beam/incoming parton indices.

⟨*Subevt expr: parton expr: TBP*⟩+≡
```
    procedure :: get_beam_index => parton_expr_get_beam_index
    procedure :: get_in_index => parton_expr_get_in_index
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine parton_expr_get_beam_index (expr, i_beam)
      class(parton_expr_t), intent(in) :: expr
      integer, dimension(:), intent(out) :: i_beam
      i_beam = expr%i_beam
    end subroutine parton_expr_get_beam_index

    subroutine parton_expr_get_in_index (expr, i_in)
      class(parton_expr_t), intent(in) :: expr
      integer, dimension(:), intent(out) :: i_in
      i_in = expr%i_in
    end subroutine parton_expr_get_in_index
```

15.3.5   Implementation for full events

This implementation contains the expressions that we can evaluate for the full event. It also contains data that pertain to the event, suitable for communication with external event formats. These data simultaneously serve as pointer targets for the variable lists hidden in the expressions (eval trees).

Squared matrix element and weight values: when reading events from file, the `ref` value is the number in the file, while the `prc` value is the number that we calculate from the momenta in the file, possibly with different parameters. When generating events the first time, or if we do not recalculate, the numbers should coincide. Furthermore, the array of `alt` values is copied from an array of alternative event records. These values should represent calculated values.

⟨*Subevt expr: public*⟩+≡
```
    public :: event_expr_t
```

⟨*Subevt expr: types*⟩+≡
```
    type, extends (subevt_expr_t) :: event_expr_t
       logical :: has_reweight = .false.
       logical :: has_analysis = .false.
       type(eval_tree_t) :: reweight
       type(eval_tree_t) :: analysis
       logical :: has_id = .false.
       type(string_t) :: id
       logical :: has_num_id = .false.
       integer :: num_id = 0
       logical :: has_index = .false.
       integer :: index = 0
       logical :: has_sqme_ref = .false.
       real(default) :: sqme_ref = 0
       logical :: has_sqme_prc = .false.
       real(default) :: sqme_prc = 0
```

```
      logical :: has_weight_ref = .false.
      real(default) :: weight_ref = 0
      logical :: has_weight_prc = .false.
      real(default) :: weight_prc = 0
      logical :: has_excess_prc = .false.
      real(default) :: excess_prc = 0
      integer :: n_alt = 0
      logical :: has_sqme_alt = .false.
      real(default), dimension(:), allocatable :: sqme_alt
      logical :: has_weight_alt = .false.
      real(default), dimension(:), allocatable :: weight_alt
    contains
    ⟨Subevt expr: event expr: TBP⟩
    end type event_expr_t
```

Finalizer for the expressions.

⟨*Subevt expr: event expr: TBP*⟩≡
```
    procedure :: final => event_expr_final
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_final (object)
      class(event_expr_t), intent(inout) :: object
      call object%base_final ()
      if (object%has_reweight) then
         call eval_tree_final (object%reweight)
      end if
      if (object%has_analysis) then
         call eval_tree_final (object%analysis)
      end if
    end subroutine event_expr_final
```

Output: continue writing the active expressions, after the common selection expression.

Note: the `prefix` argument is declared in the `write` method of the `subevt_t` base type. Here, it is unused.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: write => event_expr_write
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_write (object, unit, prefix)
      class(event_expr_t), intent(in) :: object
      integer, intent(in), optional :: unit
      character(*), intent(in), optional :: prefix
      integer :: u
      u = output_unit (unit)
      call object%base_write (u)
      if (object%subevt_filled) then
         if (object%has_reweight) then
            call write_separator (u)
            write (u, "(1x,A)")  "Reweighting expression:"
            call write_separator (u)
            call eval_tree_write (object%reweight, u)
         end if
         if (object%has_analysis) then
```

```
            call write_separator (u)
            write (u, "(1x,A)")  "Analysis expression:"
            call write_separator (u)
            call eval_tree_write (object%analysis, u)
         end if
      end if
    end subroutine event_expr_write
```

Initializer. This is required only for the `sqme_alt` and `weight_alt` arrays.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: init => event_expr_init
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_init (expr, n_alt)
      class(event_expr_t), intent(out) :: expr
      integer, intent(in), optional :: n_alt
      if (present (n_alt)) then
         expr%n_alt = n_alt
         allocate (expr%sqme_alt (n_alt), source = 0._default)
         allocate (expr%weight_alt (n_alt), source = 0._default)
      end if
    end subroutine event_expr_init
```

Define variables. We have the variables of the base type plus specific variables for full events. There is the event index.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: setup_vars => event_expr_setup_vars
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_setup_vars (expr, sqrts)
      class(event_expr_t), intent(inout), target :: expr
      real(default), intent(in) :: sqrts
      call expr%base_setup_vars (sqrts)
      call var_list_append_string_ptr (expr%var_list, &
           var_str ("$process_id"), expr%id, &
           is_known = expr%has_id, &
           locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_int_ptr (expr%var_list, &
           var_str ("process_num_id"), expr%num_id, &
           is_known = expr%has_num_id, &
           locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_real_ptr (expr%var_list, &
           var_str ("sqme"), expr%sqme_prc, &
           is_known = expr%has_sqme_prc, &
           locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_real_ptr (expr%var_list, &
           var_str ("sqme_ref"), expr%sqme_ref, &
           is_known = expr%has_sqme_ref, &
           locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_int_ptr (expr%var_list, &
           var_str ("event_index"), expr%index, &
           is_known = expr%has_index, &
           locked = .true., verbose = .false., intrinsic = .true.)
      call var_list_append_real_ptr (expr%var_list, &
```

1955

```
            var_str ("event_weight"), expr%weight_prc, &
            is_known = expr%has_weight_prc, &
            locked = .true., verbose = .false., intrinsic = .true.)
       call var_list_append_real_ptr (expr%var_list, &
            var_str ("event_weight_ref"), expr%weight_ref, &
            is_known = expr%has_weight_ref, &
            locked = .true., verbose = .false., intrinsic = .true.)
       call var_list_append_real_ptr (expr%var_list, &
            var_str ("event_excess"), expr%excess_prc, &
            is_known = expr%has_excess_prc, &
            locked = .true., verbose = .false., intrinsic = .true.)
     end subroutine event_expr_setup_vars
```

Compile the analysis expression. If the pointer is disassociated, there is no expression.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: setup_analysis => event_expr_setup_analysis
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_setup_analysis (expr, pn_analysis)
      class(event_expr_t), intent(inout), target :: expr
      type(parse_node_t), intent(in), pointer :: pn_analysis
      if (associated (pn_analysis)) then
         call eval_tree_init_lexpr (expr%analysis, &
              pn_analysis, expr%var_list, expr%subevt_t)
         expr%has_analysis = .true.
      end if
    end subroutine event_expr_setup_analysis
```

Compile the reweight expression.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: setup_reweight => event_expr_setup_reweight
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_setup_reweight (expr, pn_reweight)
      class(event_expr_t), intent(inout), target :: expr
      type(parse_node_t), intent(in), pointer :: pn_reweight
      if (associated (pn_reweight)) then
         call eval_tree_init_expr (expr%reweight, &
              pn_reweight, expr%var_list, expr%subevt_t)
         expr%has_reweight = .true.
      end if
    end subroutine event_expr_setup_reweight
```

Store the string or numeric process ID. This should be done during initialization.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: set_process_id => event_expr_set_process_id
    procedure :: set_process_num_id => event_expr_set_process_num_id
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_set_process_id (expr, id)
      class(event_expr_t), intent(inout) :: expr
      type(string_t), intent(in) :: id
```

```
      expr%id = id
      expr%has_id = .true.
    end subroutine event_expr_set_process_id

    subroutine event_expr_set_process_num_id (expr, num_id)
      class(event_expr_t), intent(inout) :: expr
      integer, intent(in) :: num_id
      expr%num_id = num_id
      expr%has_num_id = .true.
    end subroutine event_expr_set_process_num_id
```

Reset / set the data that pertain to a particular event.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
    procedure :: reset => event_expr_reset
    procedure :: set => event_expr_set
```

⟨*Subevt expr: procedures*⟩+≡
```
    subroutine event_expr_reset (expr)
      class(event_expr_t), intent(inout) :: expr
      call expr%base_reset ()
      expr%has_sqme_ref = .false.
      expr%has_sqme_prc = .false.
      expr%has_sqme_alt = .false.
      expr%has_weight_ref = .false.
      expr%has_weight_prc = .false.
      expr%has_weight_alt = .false.
      expr%has_excess_prc = .false.
    end subroutine event_expr_reset

    subroutine event_expr_set (expr, &
         weight_ref, weight_prc, weight_alt, &
         excess_prc, &
         sqme_ref, sqme_prc, sqme_alt)
      class(event_expr_t), intent(inout) :: expr
      real(default), intent(in), optional :: weight_ref, weight_prc
      real(default), intent(in), optional :: excess_prc
      real(default), intent(in), optional :: sqme_ref, sqme_prc
      real(default), dimension(:), intent(in), optional :: sqme_alt, weight_alt
      if (present (sqme_ref)) then
         expr%has_sqme_ref = .true.
         expr%sqme_ref = sqme_ref
      end if
      if (present (sqme_prc)) then
         expr%has_sqme_prc = .true.
         expr%sqme_prc = sqme_prc
      end if
      if (present (sqme_alt)) then
         expr%has_sqme_alt = .true.
         expr%sqme_alt = sqme_alt
      end if
      if (present (weight_ref)) then
         expr%has_weight_ref = .true.
         expr%weight_ref = weight_ref
      end if
```

```
      if (present (weight_prc)) then
         expr%has_weight_prc = .true.
         expr%weight_prc = weight_prc
      end if
      if (present (weight_alt)) then
         expr%has_weight_alt = .true.
         expr%weight_alt = weight_alt
      end if
      if (present (excess_prc)) then
         expr%has_excess_prc = .true.
         expr%excess_prc = excess_prc
      end if
   end subroutine event_expr_set
```

Fill the event expression: take the particle data and kinematics from a `particle_set` object.

We allow the particle content to change for each event. Therefore, we set the event variables each time.

Also increment the event index; initialize it if necessary.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
   procedure :: fill_subevt => event_expr_fill_subevt
```

⟨*Subevt expr: procedures*⟩+≡
```
   subroutine event_expr_fill_subevt (expr, particle_set)
     class(event_expr_t), intent(inout) :: expr
     type(particle_set_t), intent(in) :: particle_set
     call particle_set_to_subevt (particle_set, expr%subevt_t)
     expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
     expr%n_in  = particle_set_get_n_in  (particle_set)
     expr%n_out = particle_set_get_n_out (particle_set)
     expr%n_tot = expr%n_in + expr%n_out
     expr%subevt_filled = .true.
     if (expr%has_index) then
        expr%index = expr%index + 1
     else
        expr%index = 1
        expr%has_index = .true.
     end if
   end subroutine event_expr_fill_subevt
```

Evaluate, if the event passes the selection. For absent expressions we take default values.

⟨*Subevt expr: event expr: TBP*⟩+≡
```
   procedure :: evaluate => event_expr_evaluate
```

⟨*Subevt expr: procedures*⟩+≡
```
   subroutine event_expr_evaluate (expr, passed, reweight, analysis_flag)
     class(event_expr_t), intent(inout) :: expr
     logical, intent(out) :: passed
     real(default), intent(out) :: reweight
     logical, intent(out) :: analysis_flag
     call expr%base_evaluate (passed)
     if (passed) then
```

1958

```
            if (expr%has_reweight) then
                call eval_tree_evaluate (expr%reweight)
                if (eval_tree_result_is_known (expr%reweight)) then
                   reweight = eval_tree_get_real (expr%reweight)
                else
                   call msg_error ("Evaluate reweight expression: &
                        &result undefined")
                   reweight = 0
                end if
            else
                reweight = 1
            end if
            if (expr%has_analysis) then
                call eval_tree_evaluate (expr%analysis)
                if (eval_tree_result_is_known (expr%analysis)) then
                   analysis_flag = eval_tree_get_log (expr%analysis)
                else
                   call msg_error ("Evaluate analysis expression: &
                        &result undefined")
                   analysis_flag = .false.
                end if
            else
                analysis_flag = .true.
            end if
         end if
    end subroutine event_expr_evaluate
```

15.3.6   Test

This is the master for calling self-test procedures.

⟨*Subevt expr: public*⟩+≡
```
    public :: subevt_expr_test
```

⟨*Subevt expr: tests*⟩≡
```
    subroutine subevt_expr_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Subevt expr: execute tests⟩
  end subroutine subevt_expr_test
```

Parton-event expressions

⟨*Subevt expr: execute tests*⟩≡
```
    call test (subevt_expr_1, "subevt_expr_1", &
        "parton-event expressions", &
        u, results)
```

⟨*Subevt expr: tests*⟩+≡
```
    subroutine subevt_expr_1 (u)
      integer, intent(in) :: u
      type(string_t) :: expr_text
      type(ifile_t) :: ifile
```

1959

```
type(stream_t) :: stream
type(parse_tree_t) :: pt_cuts, pt_scale, pt_fac_scale, pt_ren_scale
type(parse_tree_t) :: pt_weight
type(parse_node_t), pointer :: pn_cuts, pn_scale, pn_fac_scale, pn_ren_scale
type(parse_node_t), pointer :: pn_weight
type(os_data_t) :: os_data
type(model_list_t) :: model_list
type(model_t), pointer :: model => null ()
type(parton_expr_t), target :: expr
real(default) :: E, Ex, m
type(vector4_t), dimension(6) :: p
integer :: i, pdg
logical :: passed
real(default) :: scale, fac_scale, ren_scale, weight

write (u, "(A)")  "* Test output: subevt_expr_1"
write (u, "(A)")  "*   Purpose: Set up a subevt and associated &
     &process-specific expressions"
write (u, "(A)")

call syntax_pexpr_init ()
call syntax_model_file_init ()
call os_data_init (os_data)
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)

write (u, "(A)")  "* Expression texts"
write (u, "(A)")


expr_text = "all Pt > 100 [s]"
write (u, "(A,A)")  "cuts = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_cuts, stream, .true.)
call stream_final (stream)
pn_cuts => parse_tree_get_root_ptr (pt_cuts)

expr_text = "sqrts"
write (u, "(A,A)")  "scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_scale, stream, .true.)
call stream_final (stream)
pn_scale => parse_tree_get_root_ptr (pt_scale)

expr_text = "sqrts_hat"
write (u, "(A,A)")  "fac_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_fac_scale, stream, .true.)
```

```
call stream_final (stream)
pn_fac_scale => parse_tree_get_root_ptr (pt_fac_scale)

expr_text = "100"
write (u, "(A,A)")  "ren_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_ren_scale, stream, .true.)
call stream_final (stream)
pn_ren_scale => parse_tree_get_root_ptr (pt_ren_scale)

expr_text = "n_tot - n_in - n_out"
write (u, "(A,A)")  "weight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)
pn_weight => parse_tree_get_root_ptr (pt_weight)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize process expr"
write (u, "(A)")

call expr%setup_vars (1000._default)
call var_list_append_real (expr%var_list, var_str ("tolerance"), 0._default)
call expr%link_var_list (model_get_var_list_ptr (model))

call expr%setup_selection (pn_cuts)
call expr%setup_scales (pn_scale, pn_fac_scale, pn_ren_scale)
call expr%setup_weight (pn_weight)

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Fill subevt and evaluate expressions"
write (u, "(A)")

call subevt_init (expr%subevt_t, 6)
E = 500._default
Ex = 400._default
m = 125._default
pdg = 25
p(1) = vector4_moving (E, sqrt (E**2 - m**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - m**2), 3)
p(3) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 3)
p(4) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 3)
p(5) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 1)
p(6) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 1)
```

```
      call expr%reset ()
      do i = 1, 2
          call subevt_set_beam (expr%subevt_t, i, pdg, p(i), m**2)
      end do
      do i = 3, 4
          call subevt_set_incoming (expr%subevt_t, i, pdg, p(i), m**2)
      end do
      do i = 5, 6
          call subevt_set_outgoing (expr%subevt_t, i, pdg, p(i), m**2)
      end do
      expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
      expr%n_in = 2
      expr%n_out = 2
      expr%n_tot = 4
      expr%subevt_filled = .true.

      call expr%evaluate (passed, scale, fac_scale, ren_scale, weight)

      write (u, "(A,L1)")      "Event has passed      = ", passed
      write (u, "(A," // FMT_12 // ")")  "Scale                 = ", scale
      write (u, "(A," // FMT_12 // ")")  "Factorization scale   = ", fac_scale
      write (u, "(A," // FMT_12 // ")")  "Renormalization scale = ", ren_scale
      write (u, "(A," // FMT_12 // ")")  "Weight                = ", weight
      write (u, "(A)")

      call write_separator (u)
      call expr%write (u)
      call write_separator (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call expr%final ()

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: subevt_expr_1"

    end subroutine subevt_expr_1
```

Parton-event expressions

⟨*Subevt expr: execute tests*⟩+≡
```
    call test (subevt_expr_2, "subevt_expr_2", &
        "parton-event expressions", &
        u, results)
```
⟨*Subevt expr: tests*⟩+≡
```
    subroutine subevt_expr_2 (u)
      integer, intent(in) :: u
```

```
type(string_t) :: expr_text
type(ifile_t) :: ifile
type(stream_t) :: stream
type(parse_tree_t) :: pt_selection
type(parse_tree_t) :: pt_reweight, pt_analysis
type(parse_node_t), pointer :: pn_selection
type(parse_node_t), pointer :: pn_reweight, pn_analysis
type(os_data_t) :: os_data
type(model_list_t) :: model_list
type(model_t), pointer :: model => null ()
type(event_expr_t), target :: expr
real(default) :: E, Ex, m
type(vector4_t), dimension(6) :: p
integer :: i, pdg
logical :: passed
real(default) :: reweight
logical :: analysis_flag

write (u, "(A)")  "* Test output: subevt_expr_2"
write (u, "(A)")  "*   Purpose: Set up a subevt and associated &
     &process-specific expressions"
write (u, "(A)")

call syntax_pexpr_init ()
call syntax_model_file_init ()
call os_data_init (os_data)
call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)

write (u, "(A)")  "* Expression texts"
write (u, "(A)")


expr_text = "all Pt > 100 [s]"
write (u, "(A,A)")  "selection = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_selection, stream, .true.)
call stream_final (stream)
pn_selection => parse_tree_get_root_ptr (pt_selection)

expr_text = "n_tot - n_in - n_out"
write (u, "(A,A)")  "reweight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_reweight, stream, .true.)
call stream_final (stream)
pn_reweight => parse_tree_get_root_ptr (pt_reweight)

expr_text = "true"
write (u, "(A,A)")  "analysis = ", char (expr_text)
call ifile_clear (ifile)
```

```
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_analysis, stream, .true.)
call stream_final (stream)
pn_analysis => parse_tree_get_root_ptr (pt_analysis)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize process expr"
write (u, "(A)")

call expr%setup_vars (1000._default)
call expr%link_var_list (model_get_var_list_ptr (model))
call var_list_append_real (expr%var_list, var_str ("tolerance"), 0._default)

call expr%setup_selection (pn_selection)
call expr%setup_analysis (pn_analysis)
call expr%setup_reweight (pn_reweight)

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Fill subevt and evaluate expressions"
write (u, "(A)")

call subevt_init (expr%subevt_t, 6)
E = 500._default
Ex = 400._default
m = 125._default
pdg = 25
p(1) = vector4_moving (E, sqrt (E**2 - m**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - m**2), 3)
p(3) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 3)
p(4) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 3)
p(5) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 1)
p(6) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 1)

call expr%reset ()
do i = 1, 2
   call subevt_set_beam (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 3, 4
   call subevt_set_incoming (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 5, 6
   call subevt_set_outgoing (expr%subevt_t, i, pdg, p(i), m**2)
end do
expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
expr%n_in = 2
expr%n_out = 2
expr%n_tot = 4
```

```
      expr%subevt_filled = .true.

      call expr%evaluate (passed, reweight, analysis_flag)

      write (u, "(A,L1)")        "Event has passed      = ", passed
      write (u, "(A," // FMT_12 // ")")  "Reweighting factor    = ", reweight
      write (u, "(A,L1)")        "Analysis flag         = ", analysis_flag
      write (u, "(A)")

      call write_separator (u)
      call expr%write (u)
      call write_separator (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call expr%final ()

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: subevt_expr_2"

   end subroutine subevt_expr_2
```

15.4 Parton states

A `parton_state_t` object contains the effective kinematics and dynamics of an elementary partonic interaction, with or without the beam/structure function state included. The type is abstract and has two distinct extensions. The `isolated_state_t` extension describes the isolated elementary interaction where the `int_eff` subobject contains the complex transition amplitude, exclusive in all quantum numbers. The particle content and kinematics describe the effective partonic state. The `connected_state_t` extension contains the partonic `subevt` and the expressions for cuts and scales which use it.

In the isolated state, the effective partonic interaction may either be identical to the hard interaction, in which case it is just a pointer to the latter. Or it may involve a rearrangement of partons, in which case we allocate it explicitly and flag this by `int_is_allocated`.

The `trace` evaluator contains the absolute square of the effective transition amplitude matrix, summed over final states. It is also summed over initial states, depending on the the beam setup allows. The result is used for integration.

The `matrix` evaluator is the counterpart of `trace` which is kept exclusive in all observable quantum numbers. The `flows` evaluator is furthermore exclusive in colors, but neglecting all color interference. The `matrix` and `flows` evaluators are filled only for sampling points that become part of physical events.

Note: It would be natural to make the evaluators allocatable. However, this causes memory corruption in gfortran 4.6.3. The extra `has_XXX` flags indicate whether evaluators are active, instead.

This module contains no unit tests. The tests are covered by the `processes` module below.

⟨parton_states.f90⟩≡
  ⟨*File header*⟩
  module parton_states

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use parser
    use lorentz !NODEP!
    use subevents
    use variables
    use expressions
    use models
    use flavors
    use helicities
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use evaluators

    use beams
    use sf_base
    use process_constants
    use prc_core
    use subevt_expr

  ⟨*Standard module head*⟩

  ⟨*Parton states: public*⟩

  ⟨*Parton states: types*⟩

  contains

  ⟨*Parton states: procedures*⟩

  end module parton_states

15.4.1 Abstract base type

The common part are the evaluators, one for the trace (summed over all quantum numbers), one for the transition matrix (summed only over unobservable quantum numbers), and one for the flow distribution (transition matrix without interferences, exclusive in color flow).

⟨*Parton states: types*⟩≡
    type, abstract :: parton_state_t
       logical :: has_trace = .false.
       logical :: has_matrix = .false.

```
    logical :: has_flows = .false.
    type(evaluator_t) :: trace
    type(evaluator_t) :: matrix
    type(evaluator_t) :: flows
  contains
  ⟨Parton states: parton state: TBP⟩
  end type parton_state_t
```

The `isolated_state_t` extension contains the `sf_chain_eff` object and the (hard) effective interaction `int_eff`, separately, both implemented as a pointer. The evaluators (trace, matrix, flows) apply to the hard interaction only.

If the effective interaction differs from the hard interaction, the pointer is allocated explicitly. Analogously for `sf_chain_eff`.

⟨*Parton states: public*⟩≡
```
    public :: isolated_state_t
```

⟨*Parton states: types*⟩+≡
```
    type, extends (parton_state_t) :: isolated_state_t
      logical :: sf_chain_is_allocated = .false.
      type(sf_chain_instance_t), pointer :: sf_chain_eff => null ()
      logical :: int_is_allocated = .false.
      type(interaction_t), pointer :: int_eff => null ()
    contains
    ⟨Parton states: isolated state: TBP⟩
    end type isolated_state_t
```

The `connected_state_t` extension contains all data that enable the evaluation of observables for the effective connected state. The evaluators connect the (effective) structure-function chain and hard interaction that were kept separate in the `isolated_state_t`.

The `flows_sf` evaluator is an extended copy of the structure-function

The `expr` subobject consists of the `subevt`, a simple event record, expressions for cuts etc. which refer to this record, and a `var_list` which contains event-specific variables, linked to the process variable list. Variables used within the expressions are looked up in `var_list`.

⟨*Parton states: types*⟩+≡
```
    public :: connected_state_t
```

⟨*Parton states: types*⟩+≡
```
    type, extends (parton_state_t) :: connected_state_t
      logical :: has_flows_sf = .false.
      type(evaluator_t) :: flows_sf
      logical :: has_expr = .false.
      type(parton_expr_t) :: expr
    contains
    ⟨Parton states: connected state: TBP⟩
    end type connected_state_t
```

Output: each evaluator is written only when it is active. The `sf_chain` is only written if it is explicitly allocated.

⟨*Parton states: parton state: TBP*⟩≡
```
    procedure :: write => parton_state_write
```

1967

⟨*Parton states: procedures*⟩≡
```
    subroutine parton_state_write (state, unit, testflag)
      class(parton_state_t), intent(in) :: state
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      select type (state)
      class is (isolated_state_t)
         if (state%sf_chain_is_allocated) then
            call write_separator (u)
            call state%sf_chain_eff%write (u)
         end if
         if (state%int_is_allocated) then
            call write_separator (u)
            write (u, "(1x,A)") &
                 "Effective interaction:"
            call write_separator (u)
            call interaction_write (state%int_eff, u, testflag = testflag)
         end if
      class is (connected_state_t)
         if (state%has_flows_sf) then
            call write_separator (u)
            write (u, "(1x,A)") &
                 "Evaluator (extension of the beam evaluator &
                 &with color contractions):"
            call write_separator (u)
            call state%flows_sf%write (u, testflag = testflag)
         end if
      end select
      if (state%has_trace) then
         call write_separator (u)
         write (u, "(1x,A)") &
              "Evaluator (trace of the squared transition matrix):"
         call write_separator (u)
         call state%trace%write (u, testflag = testflag)
      end if
      if (state%has_matrix) then
         call write_separator (u)
         write (u, "(1x,A)") &
              "Evaluator (squared transition matrix):"
         call write_separator (u)
         call state%matrix%write (u, testflag = testflag)
      end if
      if (state%has_flows) then
         call write_separator (u)
         write (u, "(1x,A)") &
              "Evaluator (squared color-flow matrix):"
         call write_separator (u)
         call state%flows%write (u, testflag = testflag)
      end if
      select type (state)
      class is (connected_state_t)
         if (state%has_expr) then
```

```
                call write_separator (u)
                call state%expr%write (u)
            end if
        end select
    end subroutine parton_state_write
```

Finalize interaction and evaluators, but only if allocated.

⟨*Parton states: parton state: TBP*⟩+≡

```
    procedure :: final => parton_state_final
```

⟨*Parton states: procedures*⟩+≡

```
    subroutine parton_state_final (state)
      class(parton_state_t), intent(inout) :: state
      if (state%has_flows) then
        call evaluator_final (state%flows)
        state%has_flows = .false.
      end if
      if (state%has_matrix) then
        call evaluator_final (state%matrix)
        state%has_matrix = .false.
      end if
      if (state%has_trace) then
        call evaluator_final (state%trace)
        state%has_trace = .false.
      end if
      select type (state)
      class is (connected_state_t)
        if (state%has_flows_sf) then
          call evaluator_final (state%flows_sf)
          state%has_flows_sf = .false.
        end if
        call state%expr%final ()
      class is (isolated_state_t)
        if (state%int_is_allocated) then
          call interaction_final (state%int_eff)
          deallocate (state%int_eff)
          state%int_is_allocated = .false.
        end if
        if (state%sf_chain_is_allocated) then
          call state%sf_chain_eff%final ()
        end if
      end select
    end subroutine parton_state_final
```

15.4.2   Common Initialization

Initialize the isolated parton state. In this version, the effective structure-function chain `sf_chain_eff` and the effective interaction `int_eff` both are trivial pointers to the seed structure-function chain and to the hard interaction, respectively.

⟨*Parton states: isolated state: TBP*⟩≡

```
    procedure :: init => isolated_state_init_pointers
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine isolated_state_init_pointers (state, sf_chain, int)
      class(isolated_state_t), intent(out) :: state
      type(sf_chain_instance_t), intent(in), target :: sf_chain
      type(interaction_t), intent(in), target :: int
      state%sf_chain_eff => sf_chain
      state%int_eff => int
    end subroutine isolated_state_init_pointers
```

15.4.3 Evaluator initialization: isolated state

Create an evaluator for the trace of the squared transition matrix. The trace goes over all outgoing quantum numbers. Whether we trace over incoming quantum numbers other than color, depends on the given `qn_mask_in`.

There are two options: explicitly computing the color factor table (`use_cf` false; `nc` defined), or taking the color factor table from the hard matrix element data.

⟨*Parton states: isolated state: TBP*⟩+≡
```
    procedure :: setup_square_trace => isolated_state_setup_square_trace
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine isolated_state_setup_square_trace (state, core, qn_mask_in, &
         col)
      class(isolated_state_t), intent(inout), target :: state
      class(prc_core_t), intent(in) :: core
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
      integer, dimension(:), intent(in) :: col
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
      associate (data => core%data)
        allocate (qn_mask (data%n_in + data%n_out))
        qn_mask(:data%n_in) = &
              new_quantum_numbers_mask (.false., .true., .false.) &
              .or. qn_mask_in
        qn_mask(data%n_in+1:) = &
            new_quantum_numbers_mask (.true., .true., .true.)
      if (core%use_color_factors) then
        call evaluator_init_square (state%trace, &
             state%int_eff, qn_mask, &
             data%cf_index, data%color_factors, col, nc=core%nc)
      else
        call evaluator_init_square (state%trace, &
             state%int_eff, qn_mask, nc=core%nc)
      end if
      end associate
      state%has_trace = .true.
    end subroutine isolated_state_setup_square_trace
```

Setup the evaluator for the transition matrix, exclusive in helicities where this is requested.

For all unstable final-state particles we keep polarization according to the applicable decay options. If the process is a decay itself, this applies also to the initial state.

For all polarized final-state particles, we keep polarization including off-diagonal entries. We drop helicity completely for unpolarized final-state particles.

For the initial state, if the particle has not been handled yet, we apply the provided `qn_mask_in` which communicates the beam properties.

⟨*Parton states: isolated state: TBP*⟩+≡
```
    procedure :: setup_square_matrix => isolated_state_setup_square_matrix
```
⟨*Parton states: procedures*⟩+≡
```
    subroutine isolated_state_setup_square_matrix &
        (state, core, model, qn_mask_in, col)
      class(isolated_state_t), intent(inout), target :: state
      class(prc_core_t), intent(in) :: core
      type(model_t), intent(in), target :: model
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
      integer, dimension(:), intent(in) :: col
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
      type(flavor_t), dimension(:), allocatable :: flv
      integer :: i
      logical :: helmask, helmask_hd
      associate (data => core%data)
        allocate (qn_mask (data%n_in + data%n_out))
        allocate (flv (data%n_flv))
        do i = 1, data%n_in + data%n_out
           call flavor_init (flv, data%flv_state(i,:), model)
           if ((data%n_in == 1 .or. i > data%n_in) &
                .and. any (.not. flavor_is_stable (flv))) then
              helmask = all (flavor_decays_isotropically (flv))
              helmask_hd = all (flavor_decays_diagonal (flv))
              qn_mask(i) = new_quantum_numbers_mask (.false., .true., helmask, &
                   mask_hd = helmask_hd)
           else if (i > data%n_in) then
              helmask = all (.not. flavor_is_polarized (flv))
              qn_mask(i) = new_quantum_numbers_mask (.false., .true., helmask)
           else
              qn_mask(i) = new_quantum_numbers_mask (.false., .true., .false.) &
                   .or. qn_mask_in(i)
           end if
        end do
      if (core%use_color_factors) then
         call evaluator_init_square (state%matrix, &
              state%int_eff, qn_mask, &
              data%cf_index, data%color_factors, col, nc=core%nc)
      else
         call evaluator_init_square (state%matrix, state%int_eff, &
              qn_mask, nc=core%nc)
      end if
      end associate
      state%has_matrix = .true.
    end subroutine isolated_state_setup_square_matrix
```

This procedure initializes the evaluator that computes the contributions to color flows, neglecting color interference. The incoming-particle mask can be used to sum over incoming flavor.

Helicity handling: see above.

⟨*Parton states: isolated state: TBP*⟩+≡
```
    procedure :: setup_square_flows => isolated_state_setup_square_flows
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine isolated_state_setup_square_flows (state, core, model, qn_mask_in)
      class(isolated_state_t), intent(inout), target :: state
      class(prc_core_t), intent(in) :: core
      type(model_t), intent(in), target :: model
      type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
      type(flavor_t), dimension(:), allocatable :: flv
      integer :: i
      logical :: helmask, helmask_hd
      associate (data => core%data)
        allocate (qn_mask (data%n_in + data%n_out))
        allocate (flv (data%n_flv))
        do i = 1, data%n_in + data%n_out
          call flavor_init (flv, data%flv_state(i,:), model)
          if ((data%n_in == 1 .or. i > data%n_in) &
              .and. any (.not. flavor_is_stable (flv))) then
            helmask = all (flavor_decays_isotropically (flv))
            helmask_hd = all (flavor_decays_diagonal (flv))
            qn_mask(i) = new_quantum_numbers_mask (.false., .false., helmask, &
                mask_hd = helmask_hd)
          else if (i > data%n_in) then
            helmask = all (.not. flavor_is_polarized (flv))
            qn_mask(i) = new_quantum_numbers_mask (.false., .false., helmask)
          else
            qn_mask(i) = new_quantum_numbers_mask (.false., .false., .false.) &
              .or. qn_mask_in(i)
          end if
        end do
        call evaluator_init_square (state%flows, state%int_eff, qn_mask, &
            expand_color_flows = .true.)
      end associate
      state%has_flows = .true.
    end subroutine isolated_state_setup_square_flows
```

15.4.4 Evaluator initialization: connected state

Setup a trace evaluator as a product of two evaluators (incoming state, effective interaction). In the result, all quantum numbers are summed over.

If the optional `int` interaction is provided, use this for the first factor in the convolution. Otherwise, use the final interaction of the stored `sf_chain`.

⟨*Parton states: connected state: TBP*⟩≡
```
    procedure :: setup_connected_trace => connected_state_setup_connected_trace
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_setup_connected_trace (state, isolated, int)
      class(connected_state_t), intent(inout), target :: state
      type(isolated_state_t), intent(in), target :: isolated
      type(interaction_t), intent(in), optional, target :: int
```

```
      type(quantum_numbers_mask_t) :: mask
      type(interaction_t), pointer :: src_int
      mask = new_quantum_numbers_mask (.true., .true., .true.)
      if (present (int)) then
         src_int => int
      else
         src_int => isolated%sf_chain_eff%get_out_int_ptr ()
      end if
      call evaluator_init_product &
           (state%trace, src_int, isolated%trace, mask, mask)
      state%has_trace = .true.
    end subroutine connected_state_setup_connected_trace
```

Setup a matrix evaluator as a product of two evaluators (incoming state, effective interation). In the intermediate state, color and helicity is summed over. In the final state, we keep the quantum numbers which are present in the original evaluators.

⟨*Parton states: connected state: TBP*⟩+≡
```
    procedure :: setup_connected_matrix => connected_state_setup_connected_matrix
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_setup_connected_matrix (state, isolated, int)
      class(connected_state_t), intent(inout), target :: state
      type(isolated_state_t), intent(in), target :: isolated
      type(interaction_t), intent(in), optional, target :: int
      type(quantum_numbers_mask_t) :: mask
      type(interaction_t), pointer :: src_int
      mask = new_quantum_numbers_mask (.false., .true., .true.)
      if (present (int)) then
         src_int => int
      else
         src_int => isolated%sf_chain_eff%get_out_int_ptr ()
      end if
      call evaluator_init_product &
           (state%matrix, src_int, isolated%matrix, mask)
      state%has_matrix = .true.
    end subroutine connected_state_setup_connected_matrix
```

Setup a matrix evaluator as a product of two evaluators (incoming state, effective interation). In the intermediate state, only helicity is summed over. In the final state, we keep the quantum numbers which are present in the original evaluators.

If the optional `int` interaction is provided, use this for the first factor in the convolution. Otherwise, use the final interaction of the stored `sf_chain`, after creating an intermediate interaction that includes a correlated color state. We assume that for a caller-provided `int`, this is not necessary.

⟨*Parton states: connected state: TBP*⟩+≡
```
    procedure :: setup_connected_flows => connected_state_setup_connected_flows
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_setup_connected_flows (state, isolated, int)
      class(connected_state_t), intent(inout), target :: state
      type(isolated_state_t), intent(in), target :: isolated
      type(interaction_t), intent(in), optional, target :: int
```

```
    type(quantum_numbers_mask_t) :: mask
    type(interaction_t), pointer :: src_int
    mask = new_quantum_numbers_mask (.false., .false., .true.)
    if (present (int)) then
       src_int => int
    else
       src_int => isolated%sf_chain_eff%get_out_int_ptr ()
       call evaluator_init_color_contractions (state%flows_sf, src_int)
       state%has_flows_sf = .true.
       src_int => evaluator_get_int_ptr (state%flows_sf)
    end if
    call evaluator_init_product &
         (state%flows, src_int, isolated%flows, mask)
    state%has_flows = .true.
  end subroutine connected_state_setup_connected_flows
```

15.4.5   Cuts and expressions

Set up the `subevt` that corresponds to the connected interaction. The index arrays refer to the interaction.

We assign the particles as follows: the beam particles are the first two (decay process: one) entries in the trace evaluator. The incoming partons are identified by their link to the outgoing partons of the structure-function chain. The outgoing partons are those of the trace evaluator, which include radiated partons during the structure-function chain.

⟨*Parton states: connected state: TBP*⟩+≡
```
    procedure :: setup_subevt => connected_state_setup_subevt
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_setup_subevt (state, sf_chain, f_beam, f_in, f_out)
      class(connected_state_t), intent(inout), target :: state
      type(sf_chain_instance_t), intent(in), target :: sf_chain
      type(flavor_t), dimension(:), intent(in) :: f_beam, f_in, f_out
      integer :: n_beam, n_in, n_out, n_vir, n_tot, i, j
      integer, dimension(:), allocatable :: i_beam, i_in, i_out
      integer :: sf_out_i
      type(interaction_t), pointer :: int, sf_int
      int => evaluator_get_int_ptr (state%trace)
      sf_int => sf_chain%get_out_int_ptr ()
      n_beam = size (f_beam)
      n_in = size (f_in)
      n_out = size (f_out)
      n_vir = interaction_get_n_vir (int)
      n_tot = interaction_get_n_tot (int)
      allocate (i_beam (n_beam), i_in (n_in), i_out (n_out))
      i_beam = [(i, i = 1, n_beam)]
      do j = 1, n_in
         sf_out_i = sf_chain%get_out_i (j)
         i_in(j) = interaction_find_link (int, sf_int, sf_out_i)
      end do
      i_out = [(i, i = n_vir + 1, n_tot)]
      call state%expr%setup_subevt (int, &
```

```
          i_beam, i_in, i_out, f_beam, f_in, f_out)
    state%has_expr = .true.
  end subroutine connected_state_setup_subevt
```

Initialize the variable list specific for this state/term. We insert event variables
(`sqrts_hat`) and link the process variable list. The variable list acquires pointers
to subobjects of `state`, which must therefore have a `target` attribute.

⟨*Parton states: connected state: TBP*⟩+≡
```
    procedure :: setup_var_list => connected_state_setup_var_list
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_setup_var_list (state, process_var_list, beam_data)
      class(connected_state_t), intent(inout), target :: state
      type(var_list_t), intent(in), target :: process_var_list
      type(beam_data_t), intent(in) :: beam_data
      call state%expr%setup_vars (beam_data_get_sqrts (beam_data))
      call state%expr%link_var_list (process_var_list)
    end subroutine connected_state_setup_var_list
```

Allocate the cut expression etc.

⟨*Parton states: connected state: TBP*⟩+≡
```
    procedure :: setup_expressions => connected_state_setup_expressions
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_setup_expressions (state, &
         pn_cuts, pn_scale, pn_fac_scale, pn_ren_scale, pn_weight)
      class(connected_state_t), intent(inout), target :: state
      type(parse_node_t), intent(in), pointer :: pn_cuts
      type(parse_node_t), intent(in), pointer :: pn_scale
      type(parse_node_t), intent(in), pointer :: pn_fac_scale
      type(parse_node_t), intent(in), pointer :: pn_ren_scale
      type(parse_node_t), intent(in), pointer :: pn_weight
      call state%expr%setup_selection (pn_cuts)
      call state%expr%setup_scales (pn_scale, pn_fac_scale, pn_ren_scale)
      call state%expr%setup_weight (pn_weight)
    end subroutine connected_state_setup_expressions
```

Reset the expression object: invalidate the subevt.

⟨*Parton states: connected state: TBP*⟩+≡
```
    procedure :: reset_expressions => connected_state_reset_expressions
```

⟨*Parton states: procedures*⟩+≡
```
    subroutine connected_state_reset_expressions (state)
      class(connected_state_t), intent(inout) :: state
      if (state%has_expr)  call state%expr%reset ()
    end subroutine connected_state_reset_expressions
```

15.4.6 Evaluation

Transfer momenta to the trace evaluator and fill the `subevt` with this effective
kinematics, if applicable.

Note: we may want to apply a boost for the `subevt`.

⟨*Parton states: parton state: TBP*⟩+≡
```
procedure :: receive_kinematics => parton_state_receive_kinematics
```

⟨*Parton states: procedures*⟩+≡
```
subroutine parton_state_receive_kinematics (state)
  class(parton_state_t), intent(inout), target :: state
  type(interaction_t), pointer :: int
  if (state%has_trace) then
     call evaluator_receive_momenta (state%trace)
     select type (state)
     class is (connected_state_t)
        if (state%has_expr) then
           int => evaluator_get_int_ptr (state%trace)
           call state%expr%fill_subevt (int)
        end if
     end select
  end if
end subroutine parton_state_receive_kinematics
```

Recover kinematics: We assume that the trace evaluator is filled with momenta. Send those momenta back to the sources, then fill the variables and subevent as above.

The incoming momenta of the connected state are not connected to the isolated state but to the beam interaction. Therefore, the incoming momenta within the isolated state do not become defined, yet. Instead, we reconstruct the beam (and ISR) momentum configuration.

⟨*Parton states: parton state: TBP*⟩+≡
```
procedure :: send_kinematics => parton_state_send_kinematics
```

⟨*Parton states: procedures*⟩+≡
```
subroutine parton_state_send_kinematics (state)
  class(parton_state_t), intent(inout), target :: state
  type(interaction_t), pointer :: int
  if (state%has_trace) then
     call evaluator_send_momenta (state%trace)
     select type (state)
     class is (connected_state_t)
        int => evaluator_get_int_ptr (state%trace)
        call state%expr%fill_subevt (int)
     end select
  end if
end subroutine parton_state_send_kinematics
```

Evaluate the expressions. The routine evaluates first the cut expression. If the event passes, it evaluates the other expressions. Where no expressions are defined, default values are inserted.

⟨*Parton states: connected state: TBP*⟩+≡
```
procedure :: evaluate_expressions => connected_state_evaluate_expressions
```

⟨*Parton states: procedures*⟩+≡
```
subroutine connected_state_evaluate_expressions (state, passed, &
     scale, fac_scale, ren_scale, weight)
```

1976

```
      class(connected_state_t), intent(inout) :: state
      logical, intent(out) :: passed
      real(default), intent(out) :: scale, fac_scale, ren_scale, weight
      if (state%has_expr) then
         call state%expr%evaluate (passed, scale, fac_scale, ren_scale, weight)
      end if
   end subroutine connected_state_evaluate_expressions
```

Evaluate the structure-function chain, if it is allocated explicitly. The argument
is the factorization scale.

If the chain is merely a pointer, the chain should already be evaluated at
this point.

⟨*Parton states: isolated state: TBP*⟩+≡
```
   procedure :: evaluate_sf_chain => isolated_state_evaluate_sf_chain
```

⟨*Parton states: procedures*⟩+≡
```
   subroutine isolated_state_evaluate_sf_chain (state, fac_scale)
     class(isolated_state_t), intent(inout) :: state
     real(default), intent(in) :: fac_scale
     if (state%sf_chain_is_allocated) then
        call state%sf_chain_eff%evaluate (fac_scale)
     end if
   end subroutine isolated_state_evaluate_sf_chain
```

Evaluate the trace.

⟨*Parton states: parton state: TBP*⟩+≡
```
   procedure :: evaluate_trace => parton_state_evaluate_trace
```

⟨*Parton states: procedures*⟩+≡
```
   subroutine parton_state_evaluate_trace (state)
     class(parton_state_t), intent(inout) :: state
     if (state%has_trace) then
        call state%trace%evaluate ()
     end if
   end subroutine parton_state_evaluate_trace
```

Evaluate the extra evaluators that we need for physical events.

⟨*Parton states: parton state: TBP*⟩+≡
```
   procedure :: evaluate_event_data => parton_state_evaluate_event_data
```

⟨*Parton states: procedures*⟩+≡
```
   subroutine parton_state_evaluate_event_data (state)
     class(parton_state_t), intent(inout) :: state
     select type (state)
     type is (connected_state_t)
        if (state%has_flows_sf) then
           call evaluator_receive_momenta (state%flows_sf)
           call state%flows_sf%evaluate ()
        end if
     end select
     if (state%has_matrix) then
        call evaluator_receive_momenta (state%matrix)
        call state%matrix%evaluate ()
```

```
      end if
      if (state%has_flows) then
         call evaluator_receive_momenta (state%flows)
         call state%flows%evaluate ()
      end if
   end subroutine parton_state_evaluate_event_data
```

Normalize the helicity density matrix by its trace, i.e., factor out the trace and put it into an overall normalization factor. The trace and flow evaluators are unchanged.

⟨*Parton states: parton state: TBP*⟩+≡
```
   procedure :: normalize_matrix_by_trace => &
        parton_state_normalize_matrix_by_trace
```

⟨*Parton states: procedures*⟩+≡
```
   subroutine parton_state_normalize_matrix_by_trace (state)
     class(parton_state_t), intent(inout) :: state
     if (state%has_matrix) then
        call evaluator_normalize_by_trace (state%matrix)
     end if
   end subroutine parton_state_normalize_matrix_by_trace
```

15.4.7   Accessing the state

Three functions return a pointer to the event-relevant interactions.

⟨*Parton states: parton state: TBP*⟩+≡
```
   procedure :: get_trace_int_ptr => parton_state_get_trace_int_ptr
   procedure :: get_matrix_int_ptr => parton_state_get_matrix_int_ptr
   procedure :: get_flows_int_ptr => parton_state_get_flows_int_ptr
```

⟨*Parton states: procedures*⟩+≡
```
   function parton_state_get_trace_int_ptr (state) result (ptr)
     class(parton_state_t), intent(in), target :: state
     type(interaction_t), pointer :: ptr
     if (state%has_trace) then
        ptr => evaluator_get_int_ptr (state%trace)
     else
        ptr => null ()
     end if
   end function parton_state_get_trace_int_ptr

   function parton_state_get_matrix_int_ptr (state) result (ptr)
     class(parton_state_t), intent(in), target :: state
     type(interaction_t), pointer :: ptr
     if (state%has_matrix) then
        ptr => evaluator_get_int_ptr (state%matrix)
     else
        ptr => null ()
     end if
   end function parton_state_get_matrix_int_ptr

   function parton_state_get_flows_int_ptr (state) result (ptr)
```

```
      class(parton_state_t), intent(in), target :: state
      type(interaction_t), pointer :: ptr
      if (state%has_flows) then
         ptr => evaluator_get_int_ptr (state%flows)
      else
         ptr => null ()
      end if
   end function parton_state_get_flows_int_ptr
```

Return the indices of the beam particles and the outgoing particles within the
trace (and thus, matrix and flows) evaluator, respectively.

⟨*Parton states: connected state: TBP*⟩+≡
```
   procedure :: get_beam_index => connected_state_get_beam_index
   procedure :: get_in_index => connected_state_get_in_index
```

⟨*Parton states: procedures*⟩+≡
```
   subroutine connected_state_get_beam_index (state, i_beam)
     class(connected_state_t), intent(in) :: state
     integer, dimension(:), intent(out) :: i_beam
     call state%expr%get_beam_index (i_beam)
   end subroutine connected_state_get_beam_index

   subroutine connected_state_get_in_index (state, i_in)
     class(connected_state_t), intent(in) :: state
     integer, dimension(:), intent(out) :: i_in
     call state%expr%get_in_index (i_in)
   end subroutine connected_state_get_in_index
```

15.5   Complete Elementary Processes

This module combines hard interactions, phase space, and (for scatterings)
structure functions and interfaces them to the integration module.

The process object implements the combination of a fixed beam and structure-
function setup with a number of elementary processes. The latter are called
process components. The process object represents an entity which is suppos-
edly observable. It should be meaningful to talk about the cross section of a
process.

The individual components of a process are, technically, processes them-
selves, but they may have unphysical cross sections which have to be added for
a physical result. Process components may be exclusive tree-level elementary
processes, dipole subtraction term, loop corrections, etc.

The beam and structure function setup is common to all process components.
Thus, there is only one instance of this part.

The process may be a scattering process or a decay process. In the latter
case, there are no structure functions, and the beam setup consists of a single
particle. Otherwise, the two classes are treated on the same footing.

Once a sampling point has been chosen, a process determines a set of partons
with a correlated density matrix of quantum numbers. In general, each sampling
point will generate, for each process component, one or more distinct parton

configurations. This is the `computed` state. The computed state is the subject of the multi-channel integration algorithm.

For NLO computations, it is necessary to project the computed states onto another set of parton configurations (e.g., by recombining certain pairs). This is the `observed` state. When computing partonic observables, the information is taken from the observed state.

For the purpose of event generation, we will later select one parton configuration from the observed state and collapse the correlated quantum state. This configuration is then dressed by applying parton shower, decays and hadronization. The decay chain, in particular, combines a scattering process with possible subsequent decay processes on the parton level, which are full-fledged process objects themselves.

⟨`processes.f90`⟩≡
  ⟨*File header*⟩

```
  module processes
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
```
    use system_dependencies !NODEP!
    use constants !NODEP!
```
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use unit_tests
    use md5
    use cputime
    use os_interface

    use ifiles
    use lexers
    use parser

    use lorentz !NODEP!
    use sm_qcd
    use pdg_arrays
    use subevents
    use variables
    use expressions
    use models
    use flavors
    use helicities
    use colors
    use quantum_numbers
    use state_matrices
    use polarizations
    use interactions
    use evaluators
    use particles
    use beam_structures
    use beams
    use sf_mappings
    use sf_base
    use process_constants
```

```
    use phs_base
    use phs_single
    use rng_base
    use mci_base
    use mci_midpoint
    use mci_vamp

    use vamp !NODEP!

    use prclib_interfaces
    use prc_core_def
    use process_libraries
    use prc_test

    use integration_results
    use prc_core
    use parton_states
```

⟨*Standard module head*⟩

⟨*Processes: public*⟩

⟨*Processes: parameters*⟩

⟨*Processes: types*⟩

⟨*Processes: process part types*⟩

⟨*Processes: process type*⟩

⟨*Processes: interfaces*⟩

⟨*Processes: test types*⟩

```
    contains
```

⟨*Processes: procedures*⟩

⟨*Processes: tests*⟩

```
    end module processes
```

15.5.1 The Process Object

A process object is the workspace for creating process instances for event generation. After initialization, its contents are filled by integration passes which shape the integration grids and compute cross sections. Processes are set up initially from user-level configuration data. After calculating integrals and thus developing integration grid data, the program may use a process object or a copy of it for the purpose of generating events

The process object consists of several subobjects with their specific purposes. The corresponding types are defined below. (Technically, the subobject type definitions have to come before the process type definition, but with NOWEB

magic we reverse this order here.)

The `type` determines whether we are considering a decay or a scattering process.

The `meta` object describes the process and its environment. All contents become fixed when the object is initialized.

The `config` object holds physical and technical configuration data that have been obtained during process initialization, and which are common to all process components.

The individual process components are configured in the `component` objects. These objects contain more configuration parameters and workspace, as needed for the specific process variant.

The `term` objects describe parton configurations which are technically used as phase-space points. Each process component may split into several terms with distinct kinematics and particle content. Furthermore, each term may project on a different physical state, e.g., by particle recombination. The `term` object provides the framework for this projection, for applying cuts, weight, and thus completing the process calculation.

The `beam_config` object describes the incoming particles, either the decay mother or the scattering beams. It also contains the structure-function information.

The `mci_entry` objects configure a MC input parameter set and integrator, each. The number of parameters depends on the process component and on the beam and structure-function setup.

⟨*Processes: public*⟩≡
```
    public :: process_t
```

⟨*Processes: process type*⟩≡
```
    type :: process_t
      private
      type(process_metadata_t) :: &
           meta
      type(process_config_data_t) :: &
           config
      type(process_counter_t) :: &
           counter
      type(process_component_t), dimension(:), allocatable :: &
           component
      type(process_term_t), dimension(:), allocatable :: &
           term
      type(process_beam_config_t) :: &
           beam_config
      type(process_mci_entry_t), dimension(:), allocatable :: &
           mci_entry
    contains
    ⟨*Processes: process: TBP*⟩
    end type process_t
```

This procedure is an important debugging and inspection tool; it is not used during normal operation. The process object is written to a file (identified by unit, which may also be standard output). Optional flags determine whether we show everything or just the interesting parts.

⟨*Processes: process: TBP*⟩≡

```
          procedure :: write => process_write
```
⟨*Processes: procedures*⟩≡
```
    subroutine process_write (process, screen, unit, &
         show_all, show_var_list, &
         show_os_data, &
         show_rng_factory, show_model, show_expressions, &
         show_sfchain, &
         show_equivalences, show_history, show_histories, &
         show_forest, show_x, &
         show_subevt, show_evaluators, pacify)
      class(process_t), intent(in) :: process
      logical, intent(in) :: screen
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: show_all
      logical, intent(in), optional :: show_var_list
      logical, intent(in), optional :: show_os_data
      logical, intent(in), optional :: show_rng_factory
      logical, intent(in), optional :: show_model, show_expressions
      logical, intent(in), optional :: show_sfchain
      logical, intent(in), optional :: show_equivalences
      logical, intent(in), optional :: show_history, show_histories
      logical, intent(in), optional :: show_forest, show_x
      logical, intent(in), optional :: show_subevt, show_evaluators
      logical, intent(in), optional :: pacify
      logical :: all
      logical :: var_list
      logical :: counters
      logical :: os_data
      logical :: rng_factory, model, expressions
      integer :: u, i
      u = output_unit (unit)
      if (present (show_all)) then
         all = show_all
      else
         all = .false.
      end if
      var_list = .false.
      counters = .true.
      os_data = .false.
      model = .false.
      rng_factory = .true.
      expressions = .false.
      if (present (show_var_list)) then
         all = .false.; var_list = show_var_list
      end if
      if (present (show_os_data)) then
         all = .false.; os_data = show_os_data
      end if
      if (present (show_rng_factory)) then
         all = .false.; rng_factory = show_rng_factory
      end if
      if (present (show_model)) then
         all = .false.; model = show_model
      end if
```

```fortran
if (present (show_expressions)) then
   all = .false.; expressions = show_expressions
end if
if (all) then
   var_list = .true.
   rng_factory = .true.
   model = .true.
   expressions = .true.
end if
if (screen) then
   write (msg_buffer, "(A)")  repeat ("-", 72)
   call msg_message ()
else
   call write_separator_double (u)
end if
call process%meta%write (u, var_list, screen)
if (process%meta%type == PRC_UNKNOWN) then
   call write_separator_double (u)
   return
else
   if (.not. screen)  call write_separator (u)
end if
if (screen)  return
call process%config%write &
     (u, counters, os_data, rng_factory, model, expressions)
call write_separator_double (u)
if (allocated (process%component)) then
   write (u, "(1x,A)") "Process component configuration:"
   do i = 1, size (process%component)
      call write_separator (u)
      call process%component(i)%write (u)
   end do
else
   write (u, "(1x,A)") "Process component configuration: [undefined]"
end if
call write_separator_double (u)
if (allocated (process%term)) then
   write (u, "(1x,A)") "Process term configuration:"
   do i = 1, size (process%term)
      call write_separator (u)
      call process%term(i)%write (u)
   end do
else
   write (u, "(1x,A)") "Process term configuration: [undefined]"
end if
call write_separator_double (u)
call process%beam_config%write (u)
call write_separator_double (u)
if (allocated (process%mci_entry)) then
   write (u, "(1x,A)") "Multi-channel integrator configurations:"
   do i = 1, size (process%mci_entry)
      call write_separator (u)
      write (u, "(1x,A,I0,A)")  "MCI #", i, ":"
      call process%mci_entry(i)%write (u, pacify)
```

```
          end do
       end if
       call write_separator_double (u)
    end subroutine process_write
```

Screen output. Write a short account of the process configuration and the
current results. The verbose version lists the components, the short version just
the results.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: show => process_show
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_show (object, unit, verbose)
      class(process_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u
      logical :: verb
      u = output_unit (unit)
      verb = .true.;  if (present (verbose)) verb = verbose
      if (verb) then
         call object%meta%show (u, object%config%model%get_name ())
         select case (object%meta%type)
         case (PRC_DECAY)
            write (u, "(2x,A)", advance="no")  "Computed width ="
         case (PRC_SCATTERING)
            write (u, "(2x,A)", advance="no")  "Computed cross section ="
         case default;  return
         end select
      else
         write (u, "(A)", advance="no") char (object%meta%id)
         select case (object%meta%num_id)
         case (0)
            write (u, "(':')")
         case default
            write (u, "(1x,'(',I0,')',':')") object%meta%num_id
         end select
         write (u, "(2x)", advance="no")
      end if
      write (u, "(ES14.7,1x,'+-',ES9.2)", advance="no") &
           object%get_integral_tot (), object%get_error_tot ()
      select case (object%meta%type)
      case (PRC_DECAY)
         write (u, "(1x,A)")  "GeV"
      case (PRC_SCATTERING)
         write (u, "(1x,A)")  "fb"
      end select
    end subroutine process_show
```

Finalizer. Explicitly iterate over all subobjects that may contain allocated point-
ers.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: final => process_final
```

1985

⟨*Processes: procedures*⟩+≡
```
    subroutine process_final (process)
      class(process_t), intent(inout) :: process
      integer :: i
      call process%meta%final ()
      call process%config%final ()
      if (allocated (process%component)) then
         do i = 1, size (process%component)
            call process%component(i)%final ()
         end do
      end if
      if (allocated (process%term)) then
         do i = 1, size (process%term)
            call process%term(i)%final ()
         end do
      end if
      call process%beam_config%final ()
      if (allocated (process%mci_entry)) then
         do i = 1, size (process%mci_entry)
            call process%mci_entry(i)%final ()
         end do
      end if
    end subroutine process_final
```

Process setup

Initialize a process. We need a process library, the name of the process, and a
run ID.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: init => process_init
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_init &
         (process, proc_id, run_id, lib, os_data, qcd, rng_factory, model_list)
      class(process_t), intent(out) :: process
      type(string_t), intent(in) :: proc_id
      type(string_t), intent(in) :: run_id
      type(process_library_t), intent(in), target :: lib
      type(os_data_t), intent(in) :: os_data
      type(qcd_t), intent(in) :: qcd
      class(rng_factory_t), intent(inout), allocatable :: rng_factory
      type(model_list_t), intent(inout) :: model_list
      if (.not. lib%is_active ()) then
         call msg_bug ("Process init: inactive library not handled yet")
      end if
      if (.not. lib%contains (proc_id)) then
         call msg_fatal ("Process library doesn't contain process '" &
              // char (proc_id) // "'")
         return
      end if
      associate (meta => process%meta)
        call meta%init (proc_id, run_id, lib)
        call process%config%init (meta, os_data, qcd, rng_factory, model_list)
```

```
      allocate (process%component (meta%n_components))
    end associate
  end subroutine process_init
```

Store a snapshot of a variable list. We link this snapshot to the current model instance and use it to update the model parameters in this instance. After update, synchronize the var list, i.e., fetch any new values of derived parameters.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: set_var_list => process_set_var_list
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_var_list (process, var_list)
      class(process_t), intent(inout) :: process
      type(var_list_t), intent(in) :: var_list
      call var_list_init_snapshot (process%meta%var_list, var_list)
      call var_list_set_original_pointers (process%meta%var_list, &
          model_get_var_list_ptr (process%config%model))
      call var_list_restore (process%meta%var_list)
      call model_parameters_update (process%config%model)
      call var_list_synchronize (process%meta%var_list, &
          model_get_var_list_ptr (process%config%model))
    end subroutine process_set_var_list
```

Initialize the process components, one by one, using a template for the process core object. The template is taken only for allocating the correct type; the contents are set by extracting the process entry from the library.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: init_component => process_init_component
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_init_component &
        (process, index, core_template, mci_template, phs_config_template)
      class(process_t), intent(inout) :: process
      integer, intent(in) :: index
      class(prc_core_t), intent(in), allocatable :: core_template
      class(mci_t), intent(in), allocatable :: mci_template
      class(phs_config_t), intent(in), allocatable :: phs_config_template
      call process%component(index)%init (index, &
          process%meta, process%config, &
          core_template, mci_template, phs_config_template)
      if (.not. process%component(index)%active) then
        call process%meta%deactivate_component(index)
      end if
    end subroutine process_init_component
```

Determine the process terms for each process component.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: setup_terms => process_setup_terms
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_terms (process)
      class(process_t), intent(inout) :: process
      type(model_t), pointer :: model
```

```
      integer :: i, j, k
      integer, dimension(:), allocatable :: n_entry
      integer :: n_components, n_tot
      model => process%config%model
      n_components = process%meta%n_components
      allocate (n_entry (n_components), source = 0)
      do i = 1, n_components
         associate (component => process%component(i))
           if (component%active)  n_entry(i) = component%core%get_n_terms ()
         end associate
      end do
      n_tot = sum (n_entry)
      allocate (process%term (n_tot))
      k = 0
      do i = 1, n_components
         associate (component => process%component(i))
           if (.not. component%active)  cycle
           associate (core => component%core)
             allocate (component%i_term (n_entry(i)))
             do j = 1, n_entry(i)
                component%i_term(j) = k + j
                call process%term(k+j)%init (k+j, i, j, core, model)
             end do
           end associate
         end associate
         k = k + n_entry(i)
      end do
      process%config%n_terms = n_tot
    end subroutine process_setup_terms
```

Initialize the beam setup. This is the trivial version where the incoming state of the matrix element coincides with the initial state of the process. For a scattering process, we need the c.m. energy, all other variables are set to their default values (no polarization, lab frame and c.m. frame coincide, etc.)

We assume that all components consistently describe a scattering process, i.e., two incoming particles.

Note: The current layout of the `beam_data_t` record requires that the flavor for each beam is unique. For processes with multiple flavors in the initial state, one has to set up beams explicitly. This restriction could be removed by extending the code in the `beams` module.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: setup_beams_sqrts => process_setup_beams_sqrts
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_beams_sqrts (process, sqrts, beam_structure)
      class(process_t), intent(inout) :: process
      real(default), intent(in) :: sqrts
      type(beam_structure_t), intent(in), optional :: beam_structure
      type(pdg_array_t), dimension(:,:), allocatable :: pdg_in
      integer, dimension(2) :: pdg_scattering
      type(flavor_t), dimension(2) :: flv_in
      integer :: i, i0
      allocate (pdg_in (2, process%meta%n_components))
```

```
        i0 = 0
        do i = 1, process%meta%n_components
           if (process%component(i)%active) then
              pdg_in(:,i) = process%component(i)%get_pdg_in ()
              if (i0 == 0)  i0 = i
           end if
        end do
        do i = 1, process%meta%n_components
           if (.not. process%component(i)%active) then
              pdg_in(:,i) = pdg_in(:,i0)
           end if
        end do
        if (all (pdg_array_get_length (pdg_in) == 1) .and. &
             all (pdg_in(1,:) == pdg_in(1,i0)) .and. &
             all (pdg_in(2,:) == pdg_in(2,i0))) then
           pdg_scattering = pdg_array_get (pdg_in(:,i0), 1)
           call flavor_init (flv_in, pdg_scattering, process%config%model)
           call process%beam_config%init_scattering (flv_in, sqrts, beam_structure)
        else
           call msg_fatal ("Setting up process '" // char (process%meta%id) // "':", &
                [var_str ("   ------------------------------------------"), &
                 var_str ("Inconsistent initial state. This happens if either "), &
                 var_str ("several processes with non-matching initial states "), &
                 var_str ("have been added, or for a single process with an "), &
                 var_str ("initial state flavor sum. In that case, please set beams "), &
                 var_str ("explicitly [singling out a flavor / structure function.]")])
        end if
      end subroutine process_setup_beams_sqrts
```

This is the version that applies to decay processes. The energy is the particle mass, hence no extra argument.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: setup_beams_decay => process_setup_beams_decay
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_beams_decay (process, rest_frame, beam_structure)
      class(process_t), intent(inout) :: process
      logical, intent(in), optional :: rest_frame
      type(beam_structure_t), intent(in), optional :: beam_structure
      type(pdg_array_t), dimension(:,:), allocatable :: pdg_in
      integer, dimension(1) :: pdg_decay
      type(flavor_t), dimension(1) :: flv_in
      integer :: i, i0
      allocate (pdg_in (1, process%meta%n_components))
      i0 = 0
      do i = 1, process%meta%n_components
         if (process%component(i)%active) then
            pdg_in(:,i) = process%component(i)%get_pdg_in ()
            if (i0 == 0)  i0 = i
         end if
      end do
      do i = 1, process%meta%n_components
         if (.not. process%component(i)%active) then
            pdg_in(:,i) = pdg_in(:,i0)
```

```
                 end if
             end do
             if (all (pdg_array_get_length (pdg_in) == 1) &
                   .and. all (pdg_in(1,:) == pdg_in(1,i0))) then
                pdg_decay = pdg_array_get (pdg_in(:,i0), 1)
                call flavor_init (flv_in, pdg_decay, process%config%model)
                call process%beam_config%init_decay (flv_in, rest_frame, beam_structure)
             else
                call msg_fatal ("Setting up decay '" &
                      // char (process%meta%id) // "': decaying particle not unique")
             end if
        end subroutine process_setup_beams_decay
```

We have to make sure that the masses of the various flavors in a given position
in the particle string coincide.

⟨*Processes: process: TBP*⟩+≡
```
        procedure :: check_masses => process_check_masses
```

⟨*Processes: procedures*⟩+≡
```
        subroutine process_check_masses (process)
            class(process_t), intent(in) :: process
            type(flavor_t), dimension(:), allocatable :: flv
            real(default), dimension(:), allocatable :: mass
            integer :: i, j
            do i = 1, process%meta%n_components
               if (.not. process%component(i)%active)  cycle
               associate (data => process%component(i)%core%data)
                 allocate (flv (data%n_flv), mass (data%n_flv))
                 do j = 1, data%n_in + data%n_out
                    call flavor_init (flv, data%flv_state(j,:), process%config%model)
                    mass = flavor_get_mass (flv)
                    if (any (mass /= mass(1))) then
                       call msg_fatal ("Process '" // char (process%meta%id) // "': " &
                             // "mass values in flavor combination do not coincide. ")
                    end if
                 end do
                 deallocate (flv, mass)
               end associate
            end do
        end subroutine process_check_masses
```

For some structure functions we need to get the list of initial state flavors. This
is a two-dimensional array. The first index is the beam index, the second index
is the component index. Each array element is itself a PDG array object, which
consists of the list of incoming PDG values for this beam and component.

⟨*Processes: process: TBP*⟩+≡
```
        procedure :: get_pdg_in => process_get_pdg_in
```

⟨*Processes: procedures*⟩+≡
```
        subroutine process_get_pdg_in (process, pdg_in)
          class(process_t), intent(in) :: process
          type(pdg_array_t), dimension(:,:), allocatable, intent(out) :: pdg_in
          integer :: i
```

```
      allocate (pdg_in (process%config%n_in, process%meta%n_components))
      do i = 1, process%meta%n_components
         if (process%component(i)%active) then
            pdg_in(:,i) = process%component(i)%get_pdg_in ()
         end if
      end do
    end subroutine process_get_pdg_in
```

Initialize from a complete beam setup.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: setup_beams_beam_structure => process_setup_beams_beam_structure
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_beams_beam_structure &
         (process, beam_structure, sqrts, model, decay_rest_frame)
      class(process_t), intent(inout) :: process
      type(beam_structure_t), intent(in) :: beam_structure
      real(default), intent(in) :: sqrts
      type(model_t), intent(in), target :: model
      logical, intent(in), optional :: decay_rest_frame
      if (process%get_n_in () == beam_structure%get_n_beam ()) then
         call process%beam_config%init_beam_structure &
              (beam_structure, sqrts, model, decay_rest_frame)
      else if (beam_structure%get_n_beam () == 0) then
         call msg_fatal ("Asymmetric beams: missing beam particle specification")
      else
         call msg_fatal ("Mismatch of process and beam setup (scattering/decay)")
      end if
    end subroutine process_setup_beams_beam_structure
```

Notify the user about beam setup.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: beams_startup_message => process_beams_startup_message
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_beams_startup_message (process, unit, beam_structure)
      class(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      type(beam_structure_t), intent(in), optional :: beam_structure
      call process%beam_config%startup_message (unit, beam_structure)
    end subroutine process_beams_startup_message
```

We complete the kinematics configuration after the beam setup, but before
we configure the chain of structure functions. The reason is that we need the
total energy `sqrts` for the kinematics, but the structure-function setup requires
the number of channels, which depends on the kinematics configuration. For
instance, the kinematics module may return the need for parameterizing an
s-channel resonance.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: configure_phs => process_configure_phs
```

1991

⟨*Processes: procedures*⟩+≡
```
subroutine process_configure_phs (process, rebuild, ignore_mismatch)
  class(process_t), intent(inout) :: process
  logical, intent(in), optional :: rebuild
  logical, intent(in), optional :: ignore_mismatch
  real(default) :: sqrts
  integer :: i
  sqrts = process%get_sqrts ()
  do i = 1, process%meta%n_components
    associate (component => process%component(i))
      if (component%active) then
        call component%configure_phs &
             (sqrts, process%beam_config, rebuild, ignore_mismatch)
      end if
    end associate
  end do
end subroutine process_configure_phs
```

Insert the structure-function configuration data. First allocate the storage, then insert data one by one. The third procedure declares a mapping (of the MC input parameters) for a specific channel and structure-function combination.

We take the number of channels from the corresponding entry in the `config_data` section.

Otherwise, these a simple wrapper routines. The extra level in the call tree may allow for simple addressing of multiple concurrent beam configurations, not implemented currently.

If we do not want structure functions, we simply do not call those procedures.

⟨*Processes: process: TBP*⟩+≡
```
procedure :: init_sf_chain => process_init_sf_chain
generic :: set_sf_channel => set_sf_channel_single
procedure :: set_sf_channel_single => process_set_sf_channel
generic :: set_sf_channel => set_sf_channel_array
procedure :: set_sf_channel_array => process_set_sf_channel_array
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_init_sf_chain (process, sf_config, sf_trace_file)
  class(process_t), intent(inout) :: process
  type(sf_config_t), dimension(:), intent(in) :: sf_config
  type(string_t), intent(in), optional :: sf_trace_file
  type(string_t) :: file
  if (present (sf_trace_file)) then
    if (sf_trace_file /= "") then
      file = sf_trace_file
    else
      file = process%get_id () // "_sftrace.dat"
    end if
    call process%beam_config%init_sf_chain (sf_config, file)
  else
    call process%beam_config%init_sf_chain (sf_config)
  end if
end subroutine process_init_sf_chain

subroutine process_set_sf_channel (process, c, sf_channel)
```

```
      class(process_t), intent(inout) :: process
      integer, intent(in) :: c
      type(sf_channel_t), intent(in) :: sf_channel
      call process%beam_config%set_sf_channel (c, sf_channel)
    end subroutine process_set_sf_channel

    subroutine process_set_sf_channel_array (process, sf_channel)
      class(process_t), intent(inout) :: process
      type(sf_channel_t), dimension(:), intent(in) :: sf_channel
      integer :: c
      call process%beam_config%allocate_sf_channels (size (sf_channel))
      do c = 1, size (sf_channel)
         call process%beam_config%set_sf_channel (c, sf_channel(c))
      end do
    end subroutine process_set_sf_channel_array
```

Notify about the structure-function setup.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: sf_startup_message => process_sf_startup_message
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_sf_startup_message (process, sf_string, unit)
      class(process_t), intent(in) :: process
      type(string_t), intent(in) :: sf_string
      integer, intent(in), optional :: unit
      call process%beam_config%sf_startup_message (sf_string, unit)
    end subroutine process_sf_startup_message
```

As soon as both the kinematics configuration and the structure-function setup are complete, we match parameterizations (channels) for both. The matching entries are (re)set in the component phase-space configuration, while the structure-function configuration is left intact.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: collect_channels => process_collect_channels
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_collect_channels (process, coll)
      class(process_t), intent(inout) :: process
      type(phs_channel_collection_t), intent(inout) :: coll
      integer :: i
      do i = 1, process%meta%n_components
         associate (component => process%component(i))
           if (component%active) then
              call component%collect_channels (coll)
           end if
         end associate
      end do
    end subroutine process_collect_channels
```

Independently, we should be able to check if any component doesn't contain phase-space parameters. Such a process can only be integrated if there are structure functions.

⟨*Processes: process: TBP*⟩+≡

1993

```
      procedure :: contains_trivial_component => process_contains_trivial_component
```
⟨*Processes: procedures*⟩+≡
```
    function process_contains_trivial_component (process) result (flag)
      class(process_t), intent(in) :: process
      logical :: flag
      integer :: i
      flag = .true.
      do i = 1, process%meta%n_components
         associate (component => process%component(i))
           if (component%active) then
              if (component%get_n_phs_par () == 0)  return
           end if
         end associate
      end do
      flag = .false.
    end function process_contains_trivial_component
```

Determine the MC parameter set structure and the MCI configuration for each
process component. We need data from the structure-function and phase-space
setup, so those should be complete before this is called. We also make a random-
number generator instance for each MCI group.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: setup_mci => process_setup_mci
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_setup_mci (process)
      class(process_t), intent(inout) :: process
      integer :: n_mci, i_mci
      integer :: i
      n_mci = 0
      do i = 1, process%meta%n_components
         associate (component => process%component(i))
           if (component%active .and. component%core%needs_mcset ()) then
              n_mci = n_mci + 1
              component%i_mci = n_mci
           end if
         end associate
      end do
      process%config%n_mci = n_mci
      if (.not. allocated (process%config%rng_factory)) &
           call msg_bug ("Process setup: rng factory not allocated")
      allocate (process%mci_entry (n_mci))
      i_mci = 0
      do i = 1, process%meta%n_components
         associate (component => process%component(i))
           if (component%active .and. component%core%needs_mcset ()) then
              i_mci = i_mci + 1
              associate (mci_entry => process%mci_entry(i_mci))
                call mci_entry%init (process%meta%type, &
                     i_mci, i, component, process%beam_config, &
                     process%config%rng_factory)
              end associate
           end if
```

```
        end associate
      end do
      do i_mci = 1, size (process%mci_entry)
          call process%mci_entry(i_mci)%set_parameters (process%meta%var_list)
      end do
    end subroutine process_setup_mci
```

Set cuts. This is a parse node, namely the right-hand side of the cut assignment.
When creating an instance, we compile this into an evaluation tree. The parse
node may be null.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: set_cuts => process_set_cuts
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_cuts (process, pn_cuts)
      class(process_t), intent(inout) :: process
      type(parse_node_t), intent(in), pointer :: pn_cuts
      process%config%pn_cuts => pn_cuts
    end subroutine process_set_cuts
```

Analogously for the other expressions.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: set_scale => process_set_scale
    procedure :: set_fac_scale => process_set_fac_scale
    procedure :: set_ren_scale => process_set_ren_scale
    procedure :: set_weight => process_set_weight
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_set_scale (process, pn_scale)
      class(process_t), intent(inout) :: process
      type(parse_node_t), intent(in), pointer :: pn_scale
      process%config%pn_scale => pn_scale
    end subroutine process_set_scale

    subroutine process_set_fac_scale (process, pn_fac_scale)
      class(process_t), intent(inout) :: process
      type(parse_node_t), intent(in), pointer :: pn_fac_scale
      process%config%pn_fac_scale => pn_fac_scale
    end subroutine process_set_fac_scale

    subroutine process_set_ren_scale (process, pn_ren_scale)
      class(process_t), intent(inout) :: process
      type(parse_node_t), intent(in), pointer :: pn_ren_scale
      process%config%pn_ren_scale => pn_ren_scale
    end subroutine process_set_ren_scale

    subroutine process_set_weight (process, pn_weight)
      class(process_t), intent(inout) :: process
      type(parse_node_t), intent(in), pointer :: pn_weight
      process%config%pn_weight => pn_weight
    end subroutine process_set_weight
```

1995

MD5 sum

The MD5 sum of the process object should reflect the state completely, including integration results. It is used for checking the integrity of event files. This global checksum includes checksums for the various parts. In particular, the MCI object receives a checksum that includes the configuration of all configuration parts relevant for an individual integration. This checksum is used for checking the integrity of integration grids.

We do not need MD5 sums for the process terms, since these are generated from the component definitions.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: compute_md5sum => process_compute_md5sum
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_compute_md5sum (process)
      class(process_t), intent(inout) :: process
      integer :: i
      call process%config%compute_md5sum ()
      do i = 1, process%config%n_components
         associate (component => process%component(i))
           if (component%active) then
              call component%compute_md5sum ()
           end if
         end associate
      end do
      call process%beam_config%compute_md5sum ()
      do i = 1, process%config%n_mci
         call process%mci_entry(i)%compute_md5sum &
              (process%config, process%component, process%beam_config)
      end do
    end subroutine process_compute_md5sum
```

Integration and event generation

The sampler test should just evaluate the squared matrix element `n_calls` times, discarding the results, and return. This can be done before integration, e.g., for timing estimates.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: sampler_test => process_sampler_test
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_sampler_test (process, instance, i_mci, n_calls)
      class(process_t), intent(inout) :: process
      type(process_instance_t), intent(inout), target :: instance
      integer, intent(in) :: i_mci
      integer, intent(in) :: n_calls
      call process%mci_entry(i_mci)%sampler_test (instance, n_calls)
    end subroutine process_sampler_test
```

Integrate the process, using a previously initialized process instance. We select one of the available MCI integrators by its index `i_mci` and thus integrate

over (structure functions and) phase space for the associated (group of) process component(s).

The finalizer should be called after all integration passes have been completed. It will, for instance, write a summary of the integration results.

`integrate_dummy` does a "dummy" integration in the sense that nothing is done but just empty integration results appended.

⟨*Processes: process: TBP*⟩+≡
```
procedure :: integrate => process_integrate
procedure :: final_integration => process_final_integration
procedure :: integrate_dummy => process_integrate_dummy
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_integrate (process, instance, i_mci, n_it, n_calls, &
     adapt_grids, adapt_weights, final, pacify)
  class(process_t), intent(inout) :: process
  type(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_mci
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  logical, intent(in), optional :: adapt_grids
  logical, intent(in), optional :: adapt_weights
  logical, intent(in), optional :: final, pacify
  call process%mci_entry(i_mci)%integrate (instance, n_it, n_calls, &
       adapt_grids, adapt_weights, final, pacify)
end subroutine process_integrate

subroutine process_final_integration (process, i_mci)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_mci
  call process%mci_entry(i_mci)%final_integration ()
end subroutine process_final_integration

subroutine process_integrate_dummy (process)
  class(process_t), intent(inout) :: process
  type(integration_results_t) :: results
  integer :: u_log
  u_log = logfile_unit ()
  call results%init (process%meta%type)
  call results%display_init (process%meta%type, screen = .true., unit = u_log)
  call results%new_pass ()
  call results%record (1, 0, 0._default, 0._default, 0._default)
  call results%display_final ()
end subroutine process_integrate_dummy
```

Display the final results for the sum of all components. (This is useful, obviously, only if there is more than one component.)

⟨*Processes: process: TBP*⟩+≡
```
procedure :: display_summed_results => process_display_summed_results
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_display_summed_results (process)
  class(process_t), intent(inout) :: process
  type(integration_results_t) :: results
  integer :: u_log
```

```
      u_log = logfile_unit ()
      call results%init (process%meta%type)
      call results%display_init (process%meta%type, screen = .true., unit = u_log)
      call results%new_pass ()
      call results%record (1, 0, &
           process%get_integral (), &
           process%get_error (), &
           process%get_efficiency ())
      call results%display_final ()
    end subroutine process_display_summed_results
```

Run LaTeX/Metapost to generate a ps/pdf file for the integration history. We (re)write the driver file – just in case it has been missed before – then we compile it.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: display_integration_history => &
        process_display_integration_history
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_display_integration_history &
        (process, i_mci, filename, os_data, eff_reset)
      class(process_t), intent(inout) :: process
      integer, intent(in) :: i_mci
      type(string_t), intent(in) :: filename
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: eff_reset
      call integration_results_write_driver &
          (process%mci_entry(i_mci)%results, filename, eff_reset)
      call integration_results_compile_driver &
          (process%mci_entry(i_mci)%results, filename, os_data)
    end subroutine process_display_integration_history
```

Write a complete logfile (with hardcoded name based on the process ID). We do not write internal data.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: write_logfile => process_write_logfile
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_write_logfile (process, i_mci, filename)
      class(process_t), intent(inout) :: process
      integer, intent(in) :: i_mci
      type(string_t), intent(in) :: filename
      type(time_t) :: time
      integer :: unit, u
      unit = free_unit ()
      open (unit = unit, file = char (filename), action = "write", &
            status = "replace")
      u = output_unit (unit)
      write (u, "(A)")  repeat ("#", 79)
      call process%meta%write (u, .false., .false.)
      write (u, "(A)")  repeat ("#", 79)
      write (u, "(3x,A,ES17.10)")  "Integral  = ", &
          process%mci_entry(i_mci)%get_integral ()
      write (u, "(3x,A,ES17.10)")  "Error     = ", &
```

```
      process%mci_entry(i_mci)%get_error ()
write (u, "(3x,A,ES17.10)")  "Accuracy  = ", &
     process%mci_entry(i_mci)%get_accuracy ()
write (u, "(3x,A,ES17.10)")  "Chi2      = ", &
     process%mci_entry(i_mci)%get_chi2 ()
write (u, "(3x,A,ES17.10)")  "Efficiency = ", &
     process%mci_entry(i_mci)%get_efficiency ()
call process%mci_entry(i_mci)%get_time (time, 10000)
if (time%is_known ()) then
   write (u, "(3x,A,1x,A)")  "T(10k evt) = ", char (time%to_string_dhms ())
else
   write (u, "(3x,A)")  "T(10k evt) =  [undefined]"
end if
call process%mci_entry(i_mci)%results%write (u)
write (u, "(A)")  repeat ("#", 79)
call process%mci_entry(i_mci)%results%write_chain_weights (u)
write (u, "(A)")  repeat ("#", 79)
call process%mci_entry(i_mci)%counter%write (u)
write (u, "(A)")  repeat ("#", 79)
select type (mci => process%mci_entry(i_mci)%mci)
type is (mci_midpoint_t)
   write (u, "(1x,A)")  "MC Integrator is Midpoint rule"
type is (mci_vamp_t)
   write (u, "(1x,A)")  "MC Integrator is VAMP"
   call write_separator (u)
   call mci%write_history (u)
   call write_separator (u)
   if (mci%grid_par%use_vamp_equivalences) then
      call vamp_equivalences_write (mci%equivalences, u)
   else
      write (u, "(3x,A)") "No VAMP equivalences have been used"
   end if
   call write_separator (u)
   call process%mci_entry(i_mci)%write_chain_weights (u)
class default
   write (u, "(1x,A)")  "MC Integrator: [unknown]"
end select
write (u, "(A)")  repeat ("#", 79)
call process%beam_config%data%write (u)
write (u, "(A)")  repeat ("#", 79)
if (associated (process%config%pn_cuts)) then
   write (u, "(3x,A)") "Cut expression:"
   call process%config%pn_cuts%write (u)
else
   write (u, "(3x,A)") "No cuts used."
end if
call write_separator (u)
if (associated (process%config%pn_scale)) then
   write (u, "(3x,A)") "Scale expression:"
   call process%config%pn_scale%write (u)
else
   write (u, "(3x,A)") "No scale expression was given."
end if
call write_separator (u)
```

1999

```
      if (associated (process%config%pn_fac_scale)) then
         write (u, "(3x,A)") "Factorization scale expression:"
         call process%config%pn_fac_scale%write (u)
      else
         write (u, "(3x,A)") "No factorization scale expression was given."
      end if
      call write_separator (u)
      if (associated (process%config%pn_ren_scale)) then
         write (u, "(3x,A)") "Renormalization scale expression:"
         call process%config%pn_ren_scale%write (u)
      else
         write (u, "(3x,A)") "No renormalization scale expression was given."
      end if
      call write_separator (u)
      if (associated (process%config%pn_weight)) then
         call write_separator (u)
         write (u, "(3x,A)") "Weight expression:"
         call process%config%pn_weight%write (u)
      else
         write (u, "(3x,A)") "No weight expression was given."
      end if
      write (u, "(A)")  repeat ("#", 79)
      write (u, "(1x,A)") "Summary of quantum-number states:"
      write (u, "(1x,A)")  " + sign: allowed and contributing"
      write (u, "(1x,A)")  " no +  : switched off at runtime"
      call process%write_state_summary (u)
      write (u, "(A)")  repeat ("#", 79)
      write (u, "(A)")  "Variable list:"
      call var_list_write (process%meta%var_list, u)
      write (u, "(A)")  repeat ("#", 79)
      close (u)
    end subroutine process_write_logfile
```

Display the quantum-number combinations of the process components, and their current status (allowed or switched off).

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: write_state_summary => process_write_state_summary
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_write_state_summary (process, unit)
      class(process_t), intent(in) :: process
      integer, intent(in), optional :: unit
      integer :: i, i_component, u
      u = output_unit (unit)
      do i = 1, size (process%term)
         call write_separator (u)
         i_component = process%term(i)%i_component
         if (i_component /= 0) then
            call process%term(i)%write_state_summary &
                 (process%component(i_component)%core, unit)
         end if
      end do
    end subroutine process_write_state_summary
```

2000

Prepare event generation for the specified MCI entry. This implies, in particular, checking the phase-space file.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: prepare_simulation => process_prepare_simulation
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_prepare_simulation (process, i_mci)
      class(process_t), intent(inout) :: process
      integer, intent(in) :: i_mci
      call process%mci_entry(i_mci)%prepare_simulation ()
    end subroutine process_prepare_simulation
```

Generate a weighted event. We select one of the available MCI integrators by its index `i_mci` and thus generate an event for the associated (group of) process component(s). The arguments exactly correspond to the initializer and finalizer above.

The resulting event is stored in the `prcoess_instance` object, which also holds the workspace of the integrator.

Note: The `process` object is declared `intent(inout)` because it contains the random-number state, which changes for each event. Otherwise, all volatile data are inside the `instance` object.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: generate_weighted_event => process_generate_weighted_event
    procedure :: generate_unweighted_event => process_generate_unweighted_event
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_generate_weighted_event (process, instance, i_mci)
      class(process_t), intent(inout) :: process
      type(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_mci
      call process%mci_entry(i_mci)%generate_weighted_event (instance)
    end subroutine process_generate_weighted_event

    subroutine process_generate_unweighted_event (process, instance, i_mci)
      class(process_t), intent(inout) :: process
      type(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_mci
      call process%mci_entry(i_mci)%generate_unweighted_event (instance)
    end subroutine process_generate_unweighted_event
```

This replaces the event generation methods for the situation that the process instance object has been filled by other means (i.e., reading and/or recalculating its contents). We just have to fill in missing MCI data, especially the event weight.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: recover_event => process_recover_event
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_recover_event (process, instance, i_term)
      class(process_t), intent(inout) :: process
      type(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_term
      call process%mci_entry(instance%i_mci)%recover_event (instance, i_term)
```

```
     end subroutine process_recover_event
```

Retrieve process data

Tell whether integral (and error) are known.

⟨*Processes: process: TBP*⟩+≡
```
    generic :: has_integral => has_integral_tot, has_integral_mci
    procedure :: has_integral_tot => process_has_integral_tot
    procedure :: has_integral_mci => process_has_integral_mci
```

⟨*Processes: procedures*⟩+≡
```
    function process_has_integral_mci (process, i_mci) result (flag)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      logical :: flag
      flag = process%mci_entry(i_mci)%has_integral ()
    end function process_has_integral_mci

    function process_has_integral_tot (process) result (flag)
      class(process_t), intent(in) :: process
      logical :: flag
      integer :: i
      flag = .true.
      do i = 1, size (process%mci_entry)
         flag = flag .and. process%mci_entry(i)%has_integral ()
      end do
    end function process_has_integral_tot
```

Return the current integral and error obtained by the integrator `i_mci`.

⟨*Processes: process: TBP*⟩+≡
```
    generic :: get_integral => get_integral_tot, get_integral_mci
    generic :: get_error => get_error_tot, get_error_mci
    generic :: get_efficiency => get_efficiency_tot, get_efficiency_mci
    procedure :: get_integral_tot => process_get_integral_tot
    procedure :: get_integral_mci => process_get_integral_mci
    procedure :: get_error_tot => process_get_error_tot
    procedure :: get_error_mci => process_get_error_mci
    procedure :: get_efficiency_tot => process_get_efficiency_tot
    procedure :: get_efficiency_mci => process_get_efficiency_mci
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_integral_mci (process, i_mci) result (integral)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      real(default) :: integral
      integral = process%mci_entry(i_mci)%get_integral ()
    end function process_get_integral_mci

    function process_get_error_mci (process, i_mci) result (error)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      real(default) :: error
      error = process%mci_entry(i_mci)%get_error ()
```

```
      end function process_get_error_mci

    function process_get_efficiency_mci (process, i_mci) result (efficiency)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      real(default) :: efficiency
      efficiency = process%mci_entry(i_mci)%get_efficiency ()
    end function process_get_efficiency_mci

    function process_get_integral_tot (process) result (integral)
      class(process_t), intent(in) :: process
      real(default) :: integral
      integer :: i
      integral = 0
      do i = 1, size (process%mci_entry)
         integral = integral + process%mci_entry(i)%get_integral ()
      end do
    end function process_get_integral_tot

    function process_get_error_tot (process) result (error)
      class(process_t), intent(in) :: process
      real(default) :: error
      real(default) :: variance
      integer :: i
      variance = 0
      do i = 1, size (process%mci_entry)
         variance = variance + process%mci_entry(i)%get_error () ** 2
      end do
      error = sqrt (variance)
    end function process_get_error_tot

    function process_get_efficiency_tot (process) result (efficiency)
      class(process_t), intent(in) :: process
      real(default) :: efficiency
      real(default) :: den, eff, int
      integer :: i
      den = 0
      do i = 1, size (process%mci_entry)
         int = process%get_integral (i)
         if (int > 0) then
            eff = process%mci_entry(i)%get_efficiency ()
            if (eff > 0) then
               den = den + int / eff
            else
               efficiency = 0
               return
            end if
         end if
      end do
      if (den > 0) then
         efficiency = process%get_integral () / den
      else
         efficiency = 0
      end if
```

```
      end function process_get_efficiency_tot
```

Return the MD5 sums that summarize the process component definitions. These values should be independent of parameters, beam details, expressions, etc. They can be used for checking the integrity of a process when reusing an old event file.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_md5sum_prc => process_get_md5sum_prc
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_md5sum_prc (process, i_component) result (md5sum)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_component
      character(32) :: md5sum
      if (process%component(i_component)%active) then
         md5sum = process%component(i_component)%config%get_md5sum ()
      else
         md5sum = ""
      end if
    end function process_get_md5sum_prc
```

Return the MD5 sums that summarize the state of the MCI integrators. These values should encode all process data, integration and phase space configuration, etc., and the integration results. They can thus be used for checking the integrity of an event-generation setup when reusing an old event file.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_md5sum_mci => process_get_md5sum_mci
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_md5sum_mci (process, i_mci) result (md5sum)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      character(32) :: md5sum
      md5sum = process%mci_entry(i_mci)%get_md5sum ()
    end function process_get_md5sum_mci
```

Return the MD5 sum of the process configuration. This should encode the process setup, data, and expressions, but no integration results.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_md5sum_cfg => process_get_md5sum_cfg
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_md5sum_cfg (process) result (md5sum)
      class(process_t), intent(in) :: process
      character(32) :: md5sum
      md5sum = process%config%md5sum
    end function process_get_md5sum_cfg
```

15.5.2 Metadata

This information describes the process and its environment. It is fixed upon initialization.

The `id` string is the name of the process object, as given by the user. The matrix element generator will use this string for naming Fortran procedures and types, so it should qualify as a Fortran name.

The `num_id` is meaningful if nonzero. It is used for communication with external programs or file standards which do not support string IDs.

The `run_id` string distinguishes among several runs for the same process. It identifies process instances with respect to adapted integration grids and similar run-specific data. The run ID is kept when copying processes for creating instances, however, so it does not distinguish event samples.

The `var_list` is a snapshot of the variable list, taken at the point where the process was initialized.

The `lib` pointer accesses the process library where the process definition and the process driver are located.

The `lib_index` is the index of entry in the process library that corresponds to the current process.

The `component_id` array identifies the individual process components.

The `component_description` is an array of human-readable strings that characterize the process components, for instance `a, b => c, d`.

The `active` mask array marks those components which are active. The others are skipped.

⟨*Processes: process part types*⟩≡
```
type :: process_metadata_t
  private
  integer :: type = PRC_UNKNOWN
  type(string_t) :: id
  integer :: num_id = 0
  type(string_t) :: run_id
  type(var_list_t) :: var_list
  type(process_library_t), pointer :: lib => null ()
  integer :: lib_index = 0
  integer :: n_components = 0
  type(string_t), dimension(:), allocatable :: component_id
  type(string_t), dimension(:), allocatable :: component_description
  logical, dimension(:), allocatable :: active
 contains
 ⟨Processes: process metadata: TBP⟩
end type process_metadata_t
```

The local var list is a snapshot and needs a finalizer.

⟨*Processes: process metadata: TBP*⟩≡
```
procedure :: final => process_metadata_final
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_metadata_final (meta)
  class(process_metadata_t), intent(inout) :: meta
  call var_list_final (meta%var_list, follow_link=.true.)
end subroutine process_metadata_final
```

Output: ID and run ID. We write the variable list only upon request.

⟨*Processes: process metadata: TBP*⟩+≡
```
procedure :: write => process_metadata_write
```

⟨*Processes: procedures*⟩+≡

```fortran
  subroutine process_metadata_write (meta, u, var_list, screen)
    class(process_metadata_t), intent(in) :: meta
    integer, intent(in) :: u
    logical, intent(in) :: var_list, screen
    integer :: i
    select case (meta%type)
    case (PRC_UNKNOWN)
       if (screen) then
          write (msg_buffer, "(A)") "Process [undefined]"
       else
          write (u, "(1x,A)") "Process [undefined]"
       end if
       return
    case (PRC_DECAY)
       if (screen) then
          write (msg_buffer, "(A,1x,A,A,A)") "Process [decay]:", &
               "'", char (meta%id), "'"
       else
          write (u, "(1x,A)", advance="no") "Process [decay]:"
       end if
    case (PRC_SCATTERING)
       if (screen) then
          write (msg_buffer, "(A,1x,A,A,A)") "Process [scattering]:", &
               "'", char (meta%id), "'"
       else
          write (u, "(1x,A)", advance="no") "Process [scattering]:"
       end if
    case default
       call msg_bug ("process_write: undefined process type")
    end select
    if (screen)  then
       call msg_message ()
    else
       write (u, "(1x,A,A,A)") "'", char (meta%id), "'"
    end if
    if (meta%num_id /= 0) then
       if (screen) then
          write (msg_buffer, "(2x,A,I0)") "ID (num)       = ", meta%num_id
          call msg_message ()
       else
          write (u, "(3x,A,I0)") "ID (num)       = ", meta%num_id
       end if
    end if
    if (screen) then
       if (meta%run_id /= "") then
          write (msg_buffer, "(2x,A,A,A)") "Run ID          = '", &
               char (meta%run_id), "'"
          call msg_message ()
       end if
    else
       write (u, "(3x,A,A,A)") "Run ID          = '", char (meta%run_id), "'"
    end if
    if (associated (meta%lib)) then
```

```
   if (screen) then
      write (msg_buffer, "(2x,A,A,A)")  "Library name  = '", &
           char (meta%lib%get_name ()), "'"
      call msg_message ()
   else
      write (u, "(3x,A,A,A)")  "Library name  = '", &
           char (meta%lib%get_name ()), "'"
   end if
else
   if (screen) then
      write (msg_buffer, "(2x,A)")  "Library name  = [not associated]"
      call msg_message ()
   else
      write (u, "(3x,A)")  "Library name  = [not associated]"
   end if
end if
if (screen) then
   write (msg_buffer, "(2x,A,I0)")  "Process index = ", meta%lib_index
   call msg_message ()
else
   write (u, "(3x,A,I0)")  "Process index = ", meta%lib_index
end if
if (allocated (meta%component_id)) then
   if (screen) then
      if (any (meta%active)) then
         write (msg_buffer, "(2x,A)")  "Process components:"
      else
         write (msg_buffer, "(2x,A)")  "Process components: [none]"
      end if
      call msg_message ()
   else
      write (u, "(3x,A)")  "Process components:"
   end if
   do i = 1, size (meta%component_id)
      if (.not. meta%active(i))  cycle
      if (screen) then
         write (msg_buffer, "(4x,I0,9A)")  i, ": '", &
              char (meta%component_id (i)), "':   ", &
              char (meta%component_description (i))
         call msg_message ()
      else
         write (u, "(5x,I0,9A)")  i, ": '", &
              char (meta%component_id (i)), "':   ", &
              char (meta%component_description (i))
      end if
   end do
end if
if (screen) then
   write (msg_buffer, "(A)")  repeat ("-", 72)
   call msg_message ()
else
   call write_separator (u)
end if
if (screen)  return
```

```
      if (var_list) then
         write (u, "(1x,A)")  "Variable list:"
         call write_separator (u)
         call var_list_write (meta%var_list, u)
      else
         write (u, "(1x,A)")  "Variable list: [not shown]"
      end if
   end subroutine process_metadata_write
```

Short output: list components.

⟨Processes: process metadata: TBP⟩+≡
```
    procedure :: show => process_metadata_show
```

⟨Processes: procedures⟩+≡
```
    subroutine process_metadata_show (meta, u, model_name)
      class(process_metadata_t), intent(in) :: meta
      integer, intent(in) :: u
      type(string_t), intent(in) :: model_name
      integer :: i
      select case (meta%type)
      case (PRC_UNKNOWN)
         write (u, "(A)") "Process: [undefined]"
         return
      case default
         write (u, "(A)", advance="no") "Process:"
      end select
      write (u, "(1x,A)", advance="no") char (meta%id)
      select case (meta%num_id)
      case (0)
      case default
         write (u, "(1x,'(',I0,')')", advance="no") meta%num_id
      end select
      select case (char (model_name))
      case ("")
      case default
         write (u, "(1x,'[',A,']')", advance="no")  char (model_name)
      end select
      write (u, *)
      if (allocated (meta%component_id)) then
         do i = 1, size (meta%component_id)
            if (meta%active(i)) then
               write (u, "(2x,I0,':',1x,A)")  i, &
                    char (meta%component_description (i))
            end if
         end do
      end if
    end subroutine process_metadata_show
```

Initialize.

⟨Processes: process metadata: TBP⟩+≡
```
    procedure :: init => process_metadata_init
```

⟨Processes: procedures⟩+≡
```
    subroutine process_metadata_init (meta, id, run_id, lib)
```

```
      class(process_metadata_t), intent(out) :: meta
      type(string_t), intent(in) :: id
      type(string_t), intent(in) :: run_id
      type(process_library_t), intent(in), target :: lib
      select case (lib%get_n_in (id))
      case (1);  meta%type = PRC_DECAY
      case (2);  meta%type = PRC_SCATTERING
      case default
         call msg_bug ("Process '" // char (id) // "': impossible n_in")
      end select
      meta%id = id
      meta%run_id = run_id
      meta%lib => lib
      meta%lib_index = lib%get_entry_index (id)
      meta%num_id = lib%get_num_id (id)
      call lib%get_component_list (id, meta%component_id)
      meta%n_components = size (meta%component_id)
      call lib%get_component_description_list (id, meta%component_description)
      allocate (meta%active (meta%n_components), source = .true.)
    end subroutine process_metadata_init
```

Mark a component as inactive.

⟨*Processes: process metadata: TBP*⟩+≡
```
    procedure :: deactivate_component => process_metadata_deactivate_component
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_metadata_deactivate_component (meta, i)
      class(process_metadata_t), intent(inout) :: meta
      integer, intent(in) :: i
      call msg_message ("Process component '" &
           // char (meta%component_id(i)) // "': matrix element vanishes")
      meta%active(i) = .false.
    end subroutine process_metadata_deactivate_component
```

15.5.3   Generic Configuration Data

This information concerns physical and technical properties of the process. It is fixed upon initialization, using data from the process specification and the variable list.

The number `n_in` is the number of incoming beam particles, simultaneously the number of incoming partons, 1 for a decay and 2 for a scattering process. (The number of outgoing partons may depend on the process component.)

The number `n_components` is the number of components that constitute the current process.

The number `n_terms` is the number of distinct contributions to the scattering matrix that constitute the current process. Each component may generate several terms.

The number `n_mci` is the number of independent MC integration configurations that this process uses. Distinct process components that share a MCI configuration may be combined pointwise. (Nevertheless, a given MC variable set may correspond to several "nearby" kinematical configurations.) This is also

the number of distinct sampling-function results that this process can generate. Process components that use distinct variable sets are added only once after an integration pass has completed.

The `model` pointer identifies the physics model and its parameters. This is a pointer to an external object.

The `rng_factory` component spawns independent random-number generators for use in integration, event generation, and event postprocessing.

Various `parse_node_t` objects are taken from the SINDARIN input. They encode expressions for evaluating cuts and scales. The workspaces for evaluating those expressions are set up in the `effective_state` subobjects. Note that these are really pointers, so the actual nodes are not stored inside the process object.

The `md5sum` is taken and used to verify the process configuration when re-reading data from file.

⟨*Processes: process part types*⟩+≡
```
type :: process_config_data_t
  private
  integer :: n_in = 0
  integer :: n_components = 0
  integer :: n_terms = 0
  integer :: n_mci = 0
  type(os_data_t) :: os_data
  class(rng_factory_t), allocatable :: rng_factory
  type(string_t) :: model_name
  type(model_t), pointer :: model => null ()
  type(qcd_t) :: qcd
  type(parse_node_t), pointer :: pn_cuts => null ()
  type(parse_node_t), pointer :: pn_scale => null ()
  type(parse_node_t), pointer :: pn_fac_scale => null ()
  type(parse_node_t), pointer :: pn_ren_scale => null ()
  type(parse_node_t), pointer :: pn_weight => null ()
  character(32) :: md5sum = ""
 contains
 ⟨Processes: process config data: TBP⟩
end type process_config_data_t
```

Here, we may compress the expressions for cuts etc.

⟨*Processes: process config data: TBP*⟩≡
```
procedure :: write => process_config_data_write
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_config_data_write (config, u, &
     counters, os_data, rng_factory, model, expressions)
  class(process_config_data_t), intent(in) :: config
  integer, intent(in) :: u
  logical, intent(in) :: counters
  logical, intent(in) :: os_data
  logical, intent(in) :: rng_factory
  logical, intent(in) :: model
  logical, intent(in) :: expressions
  write (u, "(1x,A)") "Configuration data:"
  if (counters) then
     write (u, "(3x,A,I0)") "Number of incoming particles = ", &
```

```fortran
           config%n_in
      write (u, "(3x,A,I0)") "Number of process components = ", &
           config%n_components
      write (u, "(3x,A,I0)") "Number of process terms      = ", &
           config%n_terms
      write (u, "(3x,A,I0)") "Number of MCI configurations = ", &
           config%n_mci
 end if
 if (os_data) then
    call os_data_write (config%os_data, u)
 end if
 if (associated (config%model)) then
    write (u, "(3x,A,A)")   "Model = ", char (config%model_name)
    if (model) then
       call write_separator (u)
       call config%model%write (u)
       call write_separator (u)
    end if
 else
    write (u, "(3x,A,A,A)")   "Model = ", char (config%model_name), &
         " [not associated]"
 end if
 call config%qcd%write (u, show_md5sum = .false.)
 if (rng_factory) then
    if (allocated (config%rng_factory)) then
       write (u, "(2x)", advance = "no")
       call config%rng_factory%write (u)
    end if
 end if
 call write_separator (u)
 if (expressions) then
    if (associated (config%pn_cuts)) then
       call write_separator (u)
       write (u, "(3x,A)") "Cut expression:"
       call config%pn_cuts%write (u)
    end if
    if (associated (config%pn_scale)) then
       call write_separator (u)
       write (u, "(3x,A)") "Scale expression:"
       call config%pn_scale%write (u)
    end if
    if (associated (config%pn_fac_scale)) then
       call write_separator (u)
       write (u, "(3x,A)") "Factorization scale expression:"
       call config%pn_fac_scale%write (u)
    end if
    if (associated (config%pn_ren_scale)) then
       call write_separator (u)
       write (u, "(3x,A)") "Renormalization scale expression:"
       call config%pn_ren_scale%write (u)
    end if
    if (associated (config%pn_weight)) then
       call write_separator (u)
       write (u, "(3x,A)") "Weight expression:"
```

```
         call config%pn_weight%write (u)
      end if
   else
      call write_separator (u)
      write (u, "(3x,A)") "Expressions (cut, scales, weight): [not shown]"
   end if
   if (config%md5sum /= "") then
      call write_separator (u)
      write (u, "(3x,A,A,A)")  "MD5 sum (config)  = '", config%md5sum, "'"
   end if
end subroutine process_config_data_write
```

Initialize. We use information from the process metadata and from the process library, given the process ID. We also store the currently active OS data set.

Using the model name that the library gives us, we try to load the model here. Reading from the global model library, we are sure that we are looking at the model associated to the process, not the selected one at the user level. However, when the mode names are the same, the models will be identical.

Instead of storing a model pointer, we create an instance which contains the current values of model parameters. The captured values are now disconnected from user-level changes and will stay for the lifetime of the process object. Allocating the model via a pointer guarantees the `target` attribute.

The RNG factory object is imported by moving the allocation.

⟨*Processes: process config data: TBP*⟩+≡
```
   procedure :: init => process_config_data_init
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_config_data_init &
     (config, meta, os_data, qcd, rng_factory, model_list)
  class(process_config_data_t), intent(out) :: config
  type(process_metadata_t), intent(in) :: meta
  type(os_data_t), intent(in) :: os_data
  type(qcd_t), intent(in) :: qcd
  class(rng_factory_t), intent(inout), allocatable :: rng_factory
  type(model_list_t), intent(inout) :: model_list
  type(string_t) :: filename
  type(model_t), pointer :: model
  config%n_in = meta%lib%get_n_in (meta%id)
  config%n_components = size (meta%component_id)
  config%os_data = os_data
  config%qcd = qcd
  call move_alloc (from = rng_factory, to = config%rng_factory)
  config%model_name = meta%lib%get_model_name (meta%id)
  if (config%model_name /= "") then
     filename = config%model_name // ".mdl"
     call model_list%read_model (config%model_name, &
          filename, config%os_data, model)
     allocate (config%model)
     call model_init_instance (config%model, model)
  end if
end subroutine process_config_data_init
```

Since the captured model is a separate object allocated via a pointer, we need a finalizer.

⟨*Processes: process config data: TBP*⟩+≡
```
    procedure :: final => process_config_data_final
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_config_data_final (config)
      class(process_config_data_t), intent(inout) :: config
      if (associated (config%model)) then
         call model_final (config%model)
         deallocate (config%model)
      end if
    end subroutine process_config_data_final
```

Compute the MD5 sum of the configuration data. This encodes, in particular, the model and the expressions for cut, scales, weight, etc. It should not contain the IDs and number of components, etc., since the MD5 sum should be useful for integrating individual components.

This is done only once. If the MD5 sum is nonempty, the calculation is skipped.

⟨*Processes: process config data: TBP*⟩+≡
```
    procedure :: compute_md5sum => process_config_data_compute_md5sum
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_config_data_compute_md5sum (config)
      class(process_config_data_t), intent(inout) :: config
      integer :: u
      if (config%md5sum == "") then
         u = free_unit ()
         open (u, status = "scratch", action = "readwrite")
         call config%write (u, counters = .false., os_data = .false., &
              rng_factory = .false., model = .true., expressions = .true.)
         rewind (u)
         config%md5sum = md5sum (u)
         close (u)
      end if
    end subroutine process_config_data_compute_md5sum
```

15.5.4 Beam configuration

The object `data` holds all details about the initial beam configuration. The allocatable array `sf` holds the structure-function configuration blocks. There are `n_strfun` entries in the structure-function chain (not counting the initial beam object). We maintain `n_channel` independent parameterizations of this chain. If this is greater than zero, we need a multi-channel sampling algorithm, where for each point one channel is selected to generate kinematics.

The number of parameters that are required for generating a structure-function chain is `n_sfpar`.

The flag `azimuthal_dependence` tells whether the process setup is symmetric about the beam axis in the c.m. system. This implies that there is no transversal beam polarization. The flag `lab_is_cm_frame` is obvious.

⟨*Processes: process part types*⟩+≡

```
type :: process_beam_config_t
   private
   type(beam_data_t) :: data
   integer :: n_strfun = 0
   integer :: n_channel = 1
   integer :: n_sfpar = 0
   type(sf_config_t), dimension(:), allocatable :: sf
   type(sf_channel_t), dimension(:), allocatable :: sf_channel
   logical :: azimuthal_dependence = .false.
   logical :: lab_is_cm_frame = .true.
   character(32) :: md5sum = ""
   logical :: sf_trace = .false.
   type(string_t) :: sf_trace_file
 contains
 ⟨Processes: process beam config: TBP⟩
end type process_beam_config_t
```

Here we write beam data only if they are actually used.

⟨*Processes: process beam config: TBP*⟩≡
```
    procedure :: write => process_beam_config_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_write (object, u)
      class(process_beam_config_t), intent(in) :: object
      integer, intent(in) :: u
      integer :: i, c
      call object%data%write (u)
      if (object%data%initialized) then
         write (u, "(3x,A,L1)")  "Azimuthal dependence   = ", &
              object%azimuthal_dependence
         write (u, "(3x,A,L1)")  "Lab frame is c.m. frame = ", &
              object%lab_is_cm_frame
         if (object%md5sum /= "") then
            write (u, "(3x,A,A,A)")  "MD5 sum (beams/strf) = '", &
                 object%md5sum, "'"
         end if
         if (allocated (object%sf)) then
            do i = 1, size (object%sf)
               call object%sf(i)%write (u)
            end do
            if (any_sf_channel_has_mapping (object%sf_channel)) then
               write (u, "(1x,A,L1)")  "Structure-function mappings per channel:"
               do c = 1, object%n_channel
                  write (u, "(3x,I0,':')", advance="no")  c
                  call object%sf_channel(c)%write (u)
               end do
            end if
         end if
      end if
    end subroutine process_beam_config_write
```

The beam data have a finalizer. We assume that there is none for the structure-function data.

⟨*Processes: process beam config: TBP*⟩+≡

```
      procedure :: final => process_beam_config_final
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_final (object)
      class(process_beam_config_t), intent(inout) :: object
      call beam_data_final (object%data)
    end subroutine process_beam_config_final
```

Initialize the beam setup with a given beam structure object.

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: init_beam_structure => process_beam_config_init_beam_structure
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_init_beam_structure &
        (beam_config, beam_structure, sqrts, model, decay_rest_frame)
      class(process_beam_config_t), intent(out) :: beam_config
      type(beam_structure_t), intent(in) :: beam_structure
      logical, intent(in), optional :: decay_rest_frame
      real(default), intent(in) :: sqrts
      type(model_t), intent(in), target :: model
      call beam_data_init_structure (beam_config%data, &
          beam_structure, sqrts, model, decay_rest_frame)
      beam_config%lab_is_cm_frame = beam_data_cm_frame (beam_config%data)
    end subroutine process_beam_config_init_beam_structure
```

Initialize the beam setup for a scattering process with specified flavor combination, other properties taken from the beam structure object (if any).

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: init_scattering => process_beam_config_init_scattering
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_init_scattering &
        (beam_config, flv_in, sqrts, beam_structure)
      class(process_beam_config_t), intent(out) :: beam_config
      type(flavor_t), dimension(2), intent(in) :: flv_in
      real(default), intent(in) :: sqrts
      type(beam_structure_t), intent(in), optional :: beam_structure
      if (present (beam_structure)) then
         if (beam_structure%polarized ()) then
            call beam_data_init_sqrts (beam_config%data, sqrts, flv_in, &
                 beam_structure%get_smatrix (), beam_structure%get_pol_f ())
         else
            call beam_data_init_sqrts (beam_config%data, sqrts, flv_in)
         end if
      else
         call beam_data_init_sqrts (beam_config%data, sqrts, flv_in)
      end if
    end subroutine process_beam_config_init_scattering
```

Initialize the beam setup for a decay process with specified flavor, other properties taken from the beam structure object (if present).

For a cascade decay, we set `rest_frame` to false, indicating a event-wise varying momentum. The beam data itself are initialized for the particle at rest.

⟨*Processes: process beam config: TBP*⟩+≡

```
       procedure :: init_decay => process_beam_config_init_decay
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_init_decay &
        (beam_config, flv_in, rest_frame, beam_structure)
      class(process_beam_config_t), intent(out) :: beam_config
      type(flavor_t), dimension(1), intent(in) :: flv_in
      logical, intent(in), optional :: rest_frame
      type(beam_structure_t), intent(in), optional :: beam_structure
      if (present (beam_structure)) then
         if (beam_structure%polarized ()) then
            call beam_data_init_decay (beam_config%data, flv_in, &
                 beam_structure%get_smatrix (), beam_structure%get_pol_f (), &
                 rest_frame = rest_frame)
         else
            call beam_data_init_decay (beam_config%data, flv_in, &
                 rest_frame = rest_frame)
         end if
      else
         call beam_data_init_decay (beam_config%data, flv_in, &
              rest_frame = rest_frame)
      end if
      beam_config%lab_is_cm_frame = beam_data_cm_frame (beam_config%data)
    end subroutine process_beam_config_init_decay
```

Print an informative message.

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: startup_message => process_beam_config_startup_message
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_startup_message &
        (beam_config, unit, beam_structure)
      class(process_beam_config_t), intent(in) :: beam_config
      integer, intent(in), optional :: unit
      type(beam_structure_t), intent(in), optional :: beam_structure
      integer :: u
      u = free_unit ()
      open (u, status="scratch", action="readwrite")
      if (present (beam_structure)) then
         call beam_structure%write (u)
      end if
      call beam_data_write (beam_config%data, u)
      rewind (u)
      do
         read (u, "(1x,A)", end=1)  msg_buffer
         call msg_message ()
      end do
1     continue
      close (u)
    end subroutine process_beam_config_startup_message
```

Allocate the structure-function array.

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: init_sf_chain => process_beam_config_init_sf_chain
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_init_sf_chain &
        (beam_config, sf_config, sf_trace_file)
      class(process_beam_config_t), intent(inout) :: beam_config
      type(sf_config_t), dimension(:), intent(in) :: sf_config
      type(string_t), intent(in), optional :: sf_trace_file
      integer :: i
      beam_config%n_strfun = size (sf_config)
      allocate (beam_config%sf (beam_config%n_strfun))
      do i = 1, beam_config%n_strfun
         associate (sf => sf_config(i))
           call beam_config%sf(i)%init (sf%i, sf%data)
           if (.not. sf%data%is_generator ()) then
              beam_config%n_sfpar = beam_config%n_sfpar + sf%data%get_n_par ()
           end if
         end associate
      end do
      if (present (sf_trace_file)) then
         beam_config%sf_trace = .true.
         beam_config%sf_trace_file = sf_trace_file
      end if
    end subroutine process_beam_config_init_sf_chain
```

Allocate the structure-function mapping channel array, given the requested number of channels.

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: allocate_sf_channels => process_beam_config_allocate_sf_channels
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_allocate_sf_channels (beam_config, n_channel)
      class(process_beam_config_t), intent(inout) :: beam_config
      integer, intent(in) :: n_channel
      beam_config%n_channel = n_channel
      call allocate_sf_channels (beam_config%sf_channel, &
           n_channel = n_channel, &
           n_strfun = beam_config%n_strfun)
    end subroutine process_beam_config_allocate_sf_channels
```

Set a structure-function mapping channel for an array of structure-function entries, for a single channel. (The default is no mapping.)

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: set_sf_channel => process_beam_config_set_sf_channel
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_beam_config_set_sf_channel (beam_config, c, sf_channel)
      class(process_beam_config_t), intent(inout) :: beam_config
      integer, intent(in) :: c
      type(sf_channel_t), intent(in) :: sf_channel
      beam_config%sf_channel(c) = sf_channel
    end subroutine process_beam_config_set_sf_channel
```

Print an informative startup message.

⟨*Processes: process beam config: TBP*⟩+≡
```
    procedure :: sf_startup_message => process_beam_config_sf_startup_message
```

⟨Processes: procedures⟩+≡

```
    subroutine process_beam_config_sf_startup_message &
         (beam_config, sf_string, unit)
      class(process_beam_config_t), intent(in) :: beam_config
      type(string_t), intent(in) :: sf_string
      integer, intent(in), optional :: unit
      if (beam_config%n_strfun > 0) then
         call msg_message ("Beam structure: " // char (sf_string), unit = unit)
         write (msg_buffer, "(A,3(1x,I0,1x,A))") &
              "Beam structure:", &
              beam_config%n_channel, "channels,", &
              beam_config%n_sfpar, "dimensions"
         call msg_message (unit = unit)
         if (beam_config%sf_trace) then
            call msg_message ("Beam structure: tracing &
                 &values in '" // char (beam_config%sf_trace_file) // "'")
         end if
      end if
    end subroutine process_beam_config_sf_startup_message
```

Return the PDF set currently in use, if any. This should be unique, so we scan the structure functions until we get a nonzero number.

(This implies that if the PDF set is not unique (e.g., proton and photon structure used together), this doesn't work correctly.)

⟨Processes: process beam config: TBP⟩+≡

```
    procedure :: get_pdf_set => process_beam_config_get_pdf_set
```

⟨Processes: procedures⟩+≡

```
    function process_beam_config_get_pdf_set (beam_config) result (pdf_set)
      class(process_beam_config_t), intent(in) :: beam_config
      integer :: pdf_set
      integer :: i
      if (allocated (beam_config%sf)) then
         do i = 1, size (beam_config%sf)
            pdf_set = beam_config%sf(i)%get_pdf_set ()
            if (pdf_set /= 0)  return
         end do
      else
         pdf_set = 0
      end if
    end function process_beam_config_get_pdf_set
```

Compute the MD5 sum for the complete beam setup. We rely on the default output of `write` to contain all relevant data.

This is done only once, when the MD5 sum is still empty.

⟨Processes: process beam config: TBP⟩+≡

```
    procedure :: compute_md5sum => process_beam_config_compute_md5sum
```

⟨Processes: procedures⟩+≡

```
    subroutine process_beam_config_compute_md5sum (beam_config)
      class(process_beam_config_t), intent(inout) :: beam_config
      integer :: u
      if (beam_config%md5sum == "") then
```

```
          u = free_unit ()
          open (u, status = "scratch", action = "readwrite")
          call beam_config%write (u)
          rewind (u)
          beam_config%md5sum = md5sum (u)
          close (u)
       end if
     end subroutine process_beam_config_compute_md5sum
```

Process call statistics

This object can record process calls, categorized by evaluation status. It is a part of the `mci_entry` component below.

⟨*Processes: types*⟩≡
```
    type :: process_counter_t
       integer :: total = 0
       integer :: failed_kinematics = 0
       integer :: failed_cuts = 0
       integer :: passed = 0
       integer :: evaluated = 0
       integer :: complete = 0
     contains
     ⟨Processes: process counter: TBP⟩
     end type process_counter_t
```

Here are the corresponding numeric codes:

⟨*Processes: parameters*⟩≡
```
    integer, parameter :: STAT_UNDEFINED = 0
    integer, parameter :: STAT_INITIAL = 1
    integer, parameter :: STAT_ACTIVATED = 2
    integer, parameter :: STAT_BEAM_MOMENTA = 3
    integer, parameter :: STAT_FAILED_KINEMATICS = 4
    integer, parameter :: STAT_SEED_KINEMATICS = 5
    integer, parameter :: STAT_HARD_KINEMATICS = 6
    integer, parameter :: STAT_EFF_KINEMATICS = 7
    integer, parameter :: STAT_FAILED_CUTS = 8
    integer, parameter :: STAT_PASSED_CUTS = 9
    integer, parameter :: STAT_EVALUATED_TRACE = 10
    integer, parameter :: STAT_EVENT_COMPLETE = 11
```

Output.

⟨*Processes: process counter: TBP*⟩≡
```
    procedure :: write => process_counter_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_counter_write (object, unit)
      class(process_counter_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (object%total > 0) then
         write (u, "(1x,A)")  "Call statistics (current run):"
```

```
      write (u, "(3x,A,I0)")  "total       = ", object%total
      write (u, "(3x,A,I0)")  "failed kin. = ", object%failed_kinematics
      write (u, "(3x,A,I0)")  "failed cuts = ", object%failed_cuts
      write (u, "(3x,A,I0)")  "passed cuts = ", object%passed
      write (u, "(3x,A,I0)")  "evaluated   = ", object%evaluated
    else
      write (u, "(1x,A)")  "Call statistics (current run): [no calls]"
    end if
  end subroutine process_counter_write
```

Reset. Just enforce default initialization.

⟨*Processes: process counter: TBP*⟩+≡
```
    procedure :: reset => process_counter_reset
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_counter_reset (counter)
      class(process_counter_t), intent(out) :: counter
    end subroutine process_counter_reset
```

We record an event according to the lowest status code greater or equal to the actual status. This is actually done by the process instance; the process object just copies the instance counter.

⟨*Processes: process counter: TBP*⟩+≡
```
    procedure :: record => process_counter_record
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_counter_record (counter, status)
      class(process_counter_t), intent(inout) :: counter
      integer, intent(in) :: status
      if (status <= STAT_FAILED_KINEMATICS) then
         counter%failed_kinematics = counter%failed_kinematics + 1
      else if (status <= STAT_FAILED_CUTS) then
         counter%failed_cuts = counter%failed_cuts + 1
      else if (status <= STAT_PASSED_CUTS) then
         counter%passed = counter%passed + 1
      else
         counter%evaluated = counter%evaluated + 1
      end if
      counter%total = counter%total + 1
    end subroutine process_counter_record
```

15.5.5   Multi-channel integration

The `process_mci_entry_t` block contains, for each process component that is integrated independently, the configuration data for its MC input parameters. Each input parameter set is handled by a `mci_t` integrator.

The MC input parameter set is broken down into the parameters required by the structure-function chain and the parameters required by the phase space of the elementary process.

The MD5 sum collects all information about the associated processes that may affect the integration. It does not contain the MCI object itself or integration results.

MC integration is organized in passes. Each pass may consist of several iterations, and for each iteration there is a number of calls. We store explicitly the values that apply to the current pass. Previous values are archived in the `results` object.

The `counter` receives the counter statistics from the associated process instance, for diagnostics.

The `results` object records results, broken down in passes and iterations.

⟨*Processes: process part types*⟩+≡
```
type :: process_mci_entry_t
   integer :: i_mci = 0
   integer, dimension(:), allocatable :: i_component
   integer :: process_type = PRC_UNKNOWN
   integer :: n_par = 0
   integer :: n_par_sf = 0
   integer :: n_par_phs = 0
   character(32) :: md5sum = ""
   integer :: pass = 0
   integer :: n_it = 0
   integer :: n_calls = 0
   logical :: activate_timer = .false.
   real(default) :: error_threshold = 0
   class(mci_t), allocatable :: mci
   type(process_counter_t) :: counter
   type(integration_results_t) :: results
 contains
 ⟨Processes: process mci entry: TBP⟩
end type process_mci_entry_t
```

Finalizer for the `mci` component.

⟨*Processes: process mci entry: TBP*⟩≡
```
procedure :: final => process_mci_entry_final
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_mci_entry_final (object)
  class(process_mci_entry_t), intent(inout) :: object
  if (allocated (object%mci))  call object%mci%final ()
end subroutine process_mci_entry_final
```

Output. Write pass/iteration information only if set (the pass index is nonzero). Write the MCI block only if it exists (for some self-tests it does not). Write results only if there are any.

⟨*Processes: process mci entry: TBP*⟩+≡
```
procedure :: write => process_mci_entry_write
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_mci_entry_write (object, unit, pacify)
  class(process_mci_entry_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A,I0)")  "Associated components = ", object%i_component
  write (u, "(3x,A,I0)")  "MC input parameters   = ", object%n_par
```

```
      write (u, "(3x,A,IO)")  "MC parameters (SF)    = ", object%n_par_sf
      write (u, "(3x,A,IO)")  "MC parameters (PHS)   = ", object%n_par_phs
      if (object%pass > 0) then
         write (u, "(3x,A,IO)")  "Current pass          = ", object%pass
         write (u, "(3x,A,IO)")  "Number of iterations  = ", object%n_it
         write (u, "(3x,A,IO)")  "Number of calls       = ", object%n_calls
      end if
      if (object%md5sum /= "") then
         write (u, "(3x,A,A,A)") "MD5 sum (components)  = '", object%md5sum, "'"
      end if
      if (allocated (object%mci)) then
         call object%mci%write (u)
      end if
      call object%counter%write (u)
      if (object%results%exist ()) then
         call object%results%write (u, suppress = pacify)
         call object%results%write_chain_weights (u)
      end if
    end subroutine process_mci_entry_write
```

Write the weights that are assigned to the chains of channels (call groves in the
`wood` phase-space implementation. This depends on the implementation and is
delegated to the `mci` component.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: write_chain_weights => process_mci_entry_write_chain_weights
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_write_chain_weights (mci_entry, unit)
      class(process_mci_entry_t), intent(in) :: mci_entry
      integer, intent(in), optional :: unit
      if (allocated (mci_entry%mci)) then
         call mci_entry%mci%write_chain_weights (unit)
      end if
    end subroutine process_mci_entry_write_chain_weights
```

Initialize. From the existing configuration, we obtain the number of channels
and the number of parameters, separately for the structure-function chain and
for the associated process component. We assume that the phase-space object
has already been configured.

The allocation of the MCI integrator with the appropriate concrete type is
the duty of the process core.

We assume that there is only one component associated with a MCI entry.
This restriction should be relaxed.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: init => process_mci_entry_init
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_init (mci_entry, &
         process_type, i_mci, i_component, component, beam_config, rng_factory)
      class(process_mci_entry_t), intent(out) :: mci_entry
      integer, intent(in) :: process_type
      integer, intent(in) :: i_mci
      integer, intent(in) :: i_component
```

```
        type(process_component_t), intent(in), target :: component
        type(process_beam_config_t), intent(in) :: beam_config
        class(rng_factory_t), intent(inout) :: rng_factory
        class(rng_t), allocatable :: rng
        associate (phs_config => component%phs_config)
          mci_entry%i_mci = i_mci
          allocate (mci_entry%i_component (1))
          mci_entry%i_component(1) = i_component
          mci_entry%n_par_sf = beam_config%n_sfpar
          mci_entry%n_par_phs = phs_config%get_n_par ()
          mci_entry%n_par = mci_entry%n_par_sf + mci_entry%n_par_phs
          mci_entry%process_type = process_type
          if (allocated (component%mci_template)) then
              allocate (mci_entry%mci, source=component%mci_template)
              call mci_entry%mci%record_index (mci_entry%i_mci)
              call mci_entry%mci%set_dimensions &
                  (mci_entry%n_par, phs_config%get_n_channel ())
              call mci_entry%mci%declare_flat_dimensions &
                  (phs_config%get_flat_dimensions ())
              if (phs_config%provides_equivalences) then
                 call mci_entry%mci%declare_equivalences &
                      (phs_config%channel, mci_entry%n_par_sf)
              end if
              if (phs_config%provides_chains) then
                 call mci_entry%mci%declare_chains (phs_config%chain)
              end if
              call rng_factory%make (rng)
              call mci_entry%mci%import_rng (rng)
          end if
        end associate
        call mci_entry%results%init (process_type)
      end subroutine process_mci_entry_init
```

Set some additional parameters.

⟨Processes: process mci entry: TBP⟩+≡
```
      procedure :: set_parameters => process_mci_entry_set_parameters
```

⟨Processes: procedures⟩+≡
```
      subroutine process_mci_entry_set_parameters (mci_entry, var_list)
        class(process_mci_entry_t), intent(inout) :: mci_entry
        type(var_list_t), intent(in) :: var_list
        real(default) :: error_threshold
        error_threshold = &
             var_list_get_rval (var_list, var_str ("error_threshold"))
        mci_entry%activate_timer = &
             var_list_get_lval (var_list, var_str ("?integration_timer"))
        call mci_entry%results%set_error_threshold (error_threshold)
      end subroutine process_mci_entry_set_parameters
```

Compute a MD5 sum that summarizes all information that could influence integration results, for the associated process components. We take the process-configuration MD5 sum which represents parameters, cuts, etc., the MD5 sums for the process component definitions and their phase space objects (which

should be configured), and the beam configuration MD5 sum. (The QCD setup is included in the process configuration data MD5 sum.)

Done only once, when the MD5 sum is still empty.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: compute_md5sum => process_mci_entry_compute_md5sum
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_compute_md5sum (mci_entry, &
        config, component, beam_config)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      type(process_config_data_t), intent(in) :: config
      type(process_component_t), dimension(:), intent(in) :: component
      type(process_beam_config_t), intent(in) :: beam_config
      type(string_t) :: buffer
      integer :: i
      if (mci_entry%md5sum == "") then
         buffer = config%md5sum // beam_config%md5sum
         do i = 1, size (component)
            if (component(i)%active) then
               buffer = buffer // component(i)%config%get_md5sum () &
                     // component(i)%md5sum_phs
            end if
         end do
         mci_entry%md5sum = md5sum (char (buffer))
      end if
      if (allocated (mci_entry%mci)) then
         call mci_entry%mci%set_md5sum (mci_entry%md5sum)
      end if
    end subroutine process_mci_entry_compute_md5sum
```

Test the MCI sampler by calling it a given number of time, discarding the results. The instance should be initialized.

The `mci_entry` is `intent(inout)` because the integrator contains the random-number state.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: sampler_test => process_mci_entry_sampler_test
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_sampler_test (mci_entry, instance, n_calls)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      type(process_instance_t), intent(inout), target :: instance
      integer, intent(in) :: n_calls
      call instance%choose_mci (mci_entry%i_mci)
      call instance%reset_counter ()
      call mci_entry%mci%sampler_test (instance, n_calls)
      mci_entry%counter = instance%get_counter ()
    end subroutine process_mci_entry_sampler_test
```

Integrate. The instance should be initialized.

The `integrate` method counts as an integration pass; the pass count is increased by one. We transfer the pass parameters (number of iterations and number of calls) to the actual integration routine.

The `mci_entry` is `intent(inout)` because the integrator contains the random-number state.

Note: The results are written to screen and to logfile. This behavior is hardcoded.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: integrate => process_mci_entry_integrate
    procedure :: final_integration => process_mci_entry_final_integration
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_integrate (mci_entry, instance, n_it, n_calls, &
         adapt_grids, adapt_weights, final, pacify)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      type(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: n_it
      integer, intent(in) :: n_calls
      logical, intent(in), optional :: adapt_grids
      logical, intent(in), optional :: adapt_weights
      logical, intent(in), optional :: final, pacify
      integer :: u_log
      u_log = logfile_unit ()
      call instance%choose_mci (mci_entry%i_mci)
      call instance%reset_counter ()
      mci_entry%pass = mci_entry%pass + 1
      mci_entry%n_it = n_it
      mci_entry%n_calls = n_calls
      if (mci_entry%pass == 1)  &
           call mci_entry%mci%startup_message (n_calls = n_calls)
      call mci_entry%mci%set_timer (active = mci_entry%activate_timer)
      call mci_entry%results%display_init &
           (mci_entry%process_type, screen = .true., unit = u_log)
      call mci_entry%results%new_pass ()
      associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
        call mci_entry%mci%add_pass (adapt_grids, adapt_weights, final)
        call mci_entry%mci%start_timer ()
        call mci_entry%mci%integrate (mci_instance, instance, n_it, &
             n_calls, mci_entry%results, pacify = pacify)
        call mci_entry%mci%stop_timer ()
        if (signal_is_pending ())  return
      end associate
      mci_entry%counter = instance%get_counter ()
      call mci_entry%results%display_pass (pacify)
    end subroutine process_mci_entry_integrate

    subroutine process_mci_entry_final_integration (mci_entry)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      call mci_entry%results%display_final ()
      call mci_entry%time_message ()
    end subroutine process_mci_entry_final_integration
```

If appropriate, issue an informative message about the expected time for an event sample.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: get_time => process_mci_entry_get_time
    procedure :: time_message => process_mci_entry_time_message
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_get_time (mci_entry, time, sample)
      class(process_mci_entry_t), intent(in) :: mci_entry
      type(time_t), intent(out) :: time
      integer, intent(in) :: sample
      real(default) :: time_last_pass, efficiency, calls
      time_last_pass = mci_entry%mci%get_time ()
      calls = mci_entry%results%get_n_calls ()
      efficiency = mci_entry%mci%get_efficiency ()
      if (time_last_pass > 0 .and. calls > 0 .and. efficiency > 0) then
         time = nint (time_last_pass / calls / efficiency * sample)
      end if
    end subroutine process_mci_entry_get_time

    subroutine process_mci_entry_time_message (mci_entry)
      class(process_mci_entry_t), intent(in) :: mci_entry
      type(time_t) :: time
      integer :: sample
      sample = 10000
      call mci_entry%get_time (time, sample)
      if (time%is_known ()) then
         call msg_message ("Time estimate for generating 10000 events: " &
              // char (time%to_string_dhms ()))
      end if
    end subroutine process_mci_entry_time_message
```

Prepare event generation. (For the test integrator, this does nothing. It is relevant for the VAMP integrator.)

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: prepare_simulation => process_mci_entry_prepare_simulation
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_prepare_simulation (mci_entry)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      call mci_entry%mci%prepare_simulation ()
    end subroutine process_mci_entry_prepare_simulation
```

Generate an event. The instance should be initialized, otherwise event generation is directed by the `mci` integrator subobject. The integrator instance is contained in a `mci_work` subobject of the process instance, which simultaneously serves as the sampler object. (We avoid the anti-aliasing rules if we assume that the sampling itself does not involve the integrator instance contained in the process instance.)

Regarding weighted events, we only take events which are valid, which means that they have valid kinematics and have passed cuts. Therefore, we have a rejection loop. For unweighted events, the unweighting routine should already take care of this.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: generate_weighted_event => &
         process_mci_entry_generate_weighted_event
    procedure :: generate_unweighted_event => &
         process_mci_entry_generate_unweighted_event
    procedure :: recover_event => process_mci_entry_recover_event
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_mci_entry_generate_weighted_event (mci_entry, instance)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      type(process_instance_t), intent(inout) :: instance
      call instance%choose_mci (mci_entry%i_mci)
      associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
        REJECTION: do
            call mci_entry%mci%generate_weighted_event (mci_instance, instance)
            if (signal_is_pending ())  return
            if (instance%is_valid ())  exit REJECTION
        end do REJECTION
      end associate
    end subroutine process_mci_entry_generate_weighted_event

    subroutine process_mci_entry_generate_unweighted_event (mci_entry, instance)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      type(process_instance_t), intent(inout) :: instance
      call instance%choose_mci (mci_entry%i_mci)
      associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
        call mci_entry%mci%generate_unweighted_event (mci_instance, instance)
      end associate
    end subroutine process_mci_entry_generate_unweighted_event

    subroutine process_mci_entry_recover_event (mci_entry, instance, i_term)
      class(process_mci_entry_t), intent(inout) :: mci_entry
      type(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_term
      integer :: channel
      mci_entry%i_mci = instance%i_mci
      channel = instance%get_channel ()
      associate (mci_instance => instance%mci_work(mci_entry%i_mci)%mci)
        call mci_instance%fetch (instance, channel)
      end associate
    end subroutine process_mci_entry_recover_event
```

Extract results.

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: has_integral => process_mci_entry_has_integral
    procedure :: get_integral => process_mci_entry_get_integral
    procedure :: get_error => process_mci_entry_get_error
    procedure :: get_accuracy => process_mci_entry_get_accuracy
    procedure :: get_chi2 => process_mci_entry_get_chi2
    procedure :: get_efficiency => process_mci_entry_get_efficiency
```

⟨*Processes: procedures*⟩+≡
```
    function process_mci_entry_has_integral (mci_entry) result (flag)
      class(process_mci_entry_t), intent(in) :: mci_entry
      logical :: flag
      flag = mci_entry%results%exist ()
    end function process_mci_entry_has_integral

    function process_mci_entry_get_integral (mci_entry) result (integral)
      class(process_mci_entry_t), intent(in) :: mci_entry
      real(default) :: integral
```

```
    integral = mci_entry%results%get_integral ()
  end function process_mci_entry_get_integral

  function process_mci_entry_get_error (mci_entry) result (error)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: error
    error = mci_entry%results%get_error ()
  end function process_mci_entry_get_error

  function process_mci_entry_get_accuracy (mci_entry) result (accuracy)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: accuracy
    accuracy = mci_entry%results%get_accuracy ()
  end function process_mci_entry_get_accuracy

  function process_mci_entry_get_chi2 (mci_entry) result (chi2)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: chi2
    chi2 = mci_entry%results%get_chi2 ()
  end function process_mci_entry_get_chi2

  function process_mci_entry_get_efficiency (mci_entry) result (efficiency)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: efficiency
    efficiency = mci_entry%results%get_efficiency ()
  end function process_mci_entry_get_efficiency
```

Return the MCI checksum. This may be the one used for configuration, but may also incorporate results, if they change the state of the integrator (adaptation).

⟨*Processes: process mci entry: TBP*⟩+≡
```
    procedure :: get_md5sum => process_mci_entry_get_md5sum
```

⟨*Processes: procedures*⟩+≡
```
    function process_mci_entry_get_md5sum (entry) result (md5sum)
      class(process_mci_entry_t), intent(in) :: entry
      character(32) :: md5sum
      md5sum = entry%mci%get_md5sum ()
    end function process_mci_entry_get_md5sum
```

15.5.6 Process Components

A process component is an individual contribution to a process (scattering or decay) which needs not be physical. The sum over all components should be physical.

The `index` indentifies this component within its parent process.

The actual process component is stored in the `core` subobject. We use a polymorphic subobject instead of an extension of `process_component_t`, because the individual entries in the array of process components can have different types. In short, `process_component_t` is a wrapper for the actual process variants.

If the `active` flag is false, we should skip this component. This happens if the associated process has vanishing matrix element.

The index array `i_term` points to the individual terms generated by this component. The indices refer to the parent process.

The index `i_mci` is the index of the MC integrator and parameter set which are associated to this process component.

⟨*Processes: process part types*⟩+≡
```
type :: process_component_t
  private
  type(process_component_def_t), pointer :: config => null ()
  integer :: index = 0
  class(prc_core_t), allocatable :: core
  logical :: active = .false.
  class(mci_t), allocatable :: mci_template
  integer, dimension(:), allocatable :: i_term
  integer :: i_mci = 0
  class(phs_config_t), allocatable :: phs_config
  character(32) :: md5sum_phs = ""
 contains
 ⟨Processes: process component: TBP⟩
end type process_component_t
```

Finalizer. The MCI template may (potentially) need a finalizer. The process configuration finalizer may include closing an open scratch file.

⟨*Processes: process component: TBP*⟩≡
```
procedure :: final => process_component_final
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_component_final (object)
  class(process_component_t), intent(inout) :: object
  if (allocated (object%mci_template)) then
     call object%mci_template%final ()
  end if
  if (allocated (object%phs_config)) then
     call object%phs_config%final ()
  end if
end subroutine process_component_final
```

The meaning of `verbose` depends on the process variant.

⟨*Processes: process component: TBP*⟩+≡
```
procedure :: write => process_component_write
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_component_write (object, unit)
  class(process_component_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  if (allocated (object%core)) then
     write (u, "(1x,A,I0)")  "Component #", object%index
     if (associated (object%config)) then
        call object%config%write (u)
        if (object%md5sum_phs /= "") then
           write (u, "(3x,A,A,A)")  "MD5 sum (phs)        = '", &
                object%md5sum_phs, "'"
```

```
         end if
      end if
      write (u, "(1x,A)") "Process core:"
      call object%core%write (u)
   else
      write (u, "(1x,A)") "Process component: [not allocated]"
   end if
   if (.not. object%active) then
      write (u, "(1x,A)") "[Inactive]"
      return
   end if
   write (u, "(1x,A)") "Referenced data:"
   if (allocated (object%i_term)) then
      write (u, "(3x,A,999(1x,I0))") "Terms                    =", &
            object%i_term
   else
      write (u, "(3x,A)") "Terms                    = [undefined]"
   end if
   if (object%i_mci /= 0) then
      write (u, "(3x,A,I0)") "MC dataset               = ", object%i_mci
   else
      write (u, "(3x,A)") "MC dataset               = [undefined]"
   end if
   if (allocated (object%phs_config)) then
      call object%phs_config%write (u)
   end if
 end subroutine process_component_write
```

Initialize the component.

⟨*Processes: process component: TBP*⟩+≡
```
     procedure :: init => process_component_init
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_component_init (component, &
        i_component, meta, config, &
        core_template, mci_template, phs_config_template)
      class(process_component_t), intent(out) :: component
      integer, intent(in) :: i_component
      type(process_metadata_t), intent(in) :: meta
      type(process_config_data_t), intent(in) :: config
      class(prc_core_t), intent(in), allocatable :: core_template
      class(mci_t), intent(in), allocatable :: mci_template
      class(phs_config_t), intent(in), allocatable :: phs_config_template
      component%index = i_component
      component%config => meta%lib%get_component_def_ptr (meta%id, i_component)
      allocate (component%core, source=core_template)
      call component%core%init (component%config%get_core_def_ptr (), &
          meta%lib, meta%id, i_component)
      component%active = component%core%has_matrix_element ()
      if (component%active) then
         if (allocated (mci_template)) &
              allocate (component%mci_template, source=mci_template)
         allocate (component%phs_config, source=phs_config_template)
         call component%phs_config%init (component%core%data, config%model)
```

```
            end if
      end subroutine process_component_init
```

Finalize the phase-space configuration.

⟨*Processes: process component: TBP*⟩+≡
```
      procedure :: configure_phs => process_component_configure_phs
```

⟨*Processes: procedures*⟩+≡
```
      subroutine process_component_configure_phs &
          (component, sqrts, beam_config, rebuild, ignore_mismatch)
        class(process_component_t), intent(inout) :: component
        real(default), intent(in) :: sqrts
        type(process_beam_config_t), intent(in) :: beam_config
        logical, intent(in), optional :: rebuild
        logical, intent(in), optional :: ignore_mismatch
        logical :: no_strfun
        no_strfun = beam_config%n_strfun == 0
        call component%phs_config%configure (sqrts, &
             azimuthal_dependence = beam_config%azimuthal_dependence, &
             sqrts_fixed = no_strfun, &
             cm_frame = beam_config%lab_is_cm_frame .and. no_strfun, &
             rebuild = rebuild, ignore_mismatch = ignore_mismatch)
        call component%phs_config%startup_message ()
      end subroutine process_component_configure_phs
```

The process component possesses two MD5 sums: the checksum of the component definition, which should be available when the component is initialized, and the phase-space MD5 sum, which is available after configuration.

⟨*Processes: process component: TBP*⟩+≡
```
      procedure :: compute_md5sum => process_component_compute_md5sum
```

⟨*Processes: procedures*⟩+≡
```
      subroutine process_component_compute_md5sum (component)
        class(process_component_t), intent(inout) :: component
        component%md5sum_phs = component%phs_config%get_md5sum ()
      end subroutine process_component_compute_md5sum
```

Match phase-space channels with structure-function channels, where applicable.
    This calls a method of the `phs_config` phase-space implementation.

⟨*Processes: process component: TBP*⟩+≡
```
      procedure :: collect_channels => process_component_collect_channels
```

⟨*Processes: procedures*⟩+≡
```
      subroutine process_component_collect_channels (component, coll)
        class(process_component_t), intent(inout) :: component
        type(phs_channel_collection_t), intent(inout) :: coll
        call component%phs_config%collect_channels (coll)
      end subroutine process_component_collect_channels
```

Return the number of phase-space parameters.

⟨*Processes: process component: TBP*⟩+≡
```
      procedure :: get_n_phs_par => process_component_get_n_phs_par
```

⟨*Processes: procedures*⟩+≡
```
    function process_component_get_n_phs_par (component) result (n_par)
      class(process_component_t), intent(in) :: component
      integer :: n_par
      n_par = component%phs_config%get_n_par ()
    end function process_component_get_n_phs_par
```

Return the incoming flavor combination as an array of PDG arrays, one for a decay or two for a scattering process.

⟨*Processes: process component: TBP*⟩+≡
```
    procedure :: get_pdg_in => process_component_get_pdg_in
```

⟨*Processes: procedures*⟩+≡
```
    function process_component_get_pdg_in (component) result (pdg_in)
      class(process_component_t), intent(in) :: component
      type(pdg_array_t), dimension(:), allocatable :: pdg_in
      type(pdg_array_t) :: pdg_tmp
      integer :: i
      associate (data => component%core%data)
        allocate (pdg_in (data%n_in))
        do i = 1, data%n_in
           pdg_tmp = data%flv_state(i,:)
           pdg_in(i) = sort_abs (pdg_tmp, unique = .true.)
        end do
      end associate
    end function process_component_get_pdg_in
```

15.5.7 Process terms

For straightforward tree-level calculations, each process component corresponds to a unique elementary interaction. However, in the case of NLO calculations with subtraction terms, a process component may split into several separate contributions to the scattering, which are qualified by interactions with distinct kinematics and particle content. We represent their configuration as `process_term_t` objects, the actual instances will be introduced below as `term_instance_t`. In any case, the process term contains an elementary interaction with a definite quantum-number and momentum content.

The index `i_term_global` identifies the term relative to the process.

The index `i_component` identifies the process component which generates this term, relative to the parent process.

The index `i_term` identifies the term relative to the process component (not the process).

The `data` subobject holds all process constants.

The number of allowed flavor/helicity/color combinations is stored as `n_allowed`. This is the total number of independent entries in the density matrix. For each combination, the index of the flavor, helicity, and color state is stored in the arrays `flv`, `hel`, and `col`, respectively.

The flag `rearrange` is true if we need to rearrange the particles of the hard interaction, to obtain the effective parton state.

The interaction `int` holds the quantum state for the (resolved) hard interaction, the parent-child relations of the particles, and their momenta. The

2032

momenta are not filled yet; this is postponed to copies of `int` which go into the process instances.

If recombination is in effect, we should allocate `int_eff` to describe the rearranged partonic state.

⟨*Processes: process part types*⟩+≡
```
type :: process_term_t
   integer :: i_term_global = 0
   integer :: i_component = 0
   integer :: i_term = 0
   integer :: n_allowed = 0
   type(process_constants_t) :: data
   real(default) :: alpha_s = 0
   integer, dimension(:), allocatable :: flv, hel, col
   logical :: rearrange = .false.
   type(interaction_t) :: int
   type(interaction_t), pointer :: int_eff => null ()
 contains
 ⟨Processes: process term: TBP⟩
end type process_term_t
```

For the output, we skip the process constants and the tables of allowed quantum numbers. Those can also be read off from the interaction object.

⟨*Processes: process term: TBP*⟩≡
```
procedure :: write => process_term_write
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_term_write (term, unit)
  class(process_term_t), intent(in) :: term
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A,I0)")  "Term #", term%i_term_global
  write (u, "(3x,A,I0)")  "Process component index     = ", &
       term%i_component
  write (u, "(3x,A,I0)")  "Term index w.r.t. component = ", &
       term%i_term
  write (u, "(3x,A,L1)")  "Rearrange partons           = ", &
       term%rearrange
  call write_separator (u)
  write (u, "(1x,A)")  "Hard interaction:"
  call write_separator (u)
  call interaction_write (term%int, u)
  if (term%rearrange) then
     call write_separator (u)
     write (u, "(1x,A)")  "Rearranged hard interaction:"
     call write_separator (u)
     call interaction_write (term%int_eff, u)
  end if
end subroutine process_term_write
```

Write an account of all quantum number states and their current status.

⟨*Processes: process term: TBP*⟩+≡
```
procedure :: write_state_summary => process_term_write_state_summary
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_term_write_state_summary (term, core, unit)
      class(process_term_t), intent(in) :: term
      class(prc_core_t), intent(in) :: core
      integer, intent(in), optional :: unit
      integer :: u, i, f, h, c
      type(state_iterator_t) :: it
      character :: sgn
      u = output_unit (unit)
      write (u, "(1x,A,I0)")  "Term #", term%i_term_global
      call state_iterator_init (it, interaction_get_state_matrix_ptr (term%int))
      do while (state_iterator_is_valid (it))
         i = state_iterator_get_me_index (it)
         f = term%flv(i)
         h = term%hel(i)
         c = term%col(i)
         if (core%is_allowed (term%i_term, f, h, c)) then
            sgn = "+"
         else
            sgn = " "
         end if
         write (u, "(1x,A1,1x,I0,2x)", advance="no")  sgn, i
         call quantum_numbers_write (state_iterator_get_quantum_numbers (it), u)
         write (u, *)
         call state_iterator_advance (it)
      end do
    end subroutine process_term_write_state_summary
```

Finalizer: the `int` and potentially `int_eff` components have a finalizer that we must call.

⟨*Processes: process term: TBP*⟩+≡
```
    procedure :: final => process_term_final
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_term_final (term)
      class(process_term_t), intent(inout) :: term
      call interaction_final (term%int)
      if (term%rearrange) then
         call interaction_final (term%int_eff)
         deallocate (term%int_eff)
      end if
    end subroutine process_term_final
```

Initialize the term. We copy the process constants from the `core` object and set up the `int` hard interaction accordingly.

The `alpha_s` value is useful for writing external event records. This is the constant value which may be overridden by a event-specific running value. If the model does not contain the strong coupling, the value is zero.

The `rearrange` part is commented out; this or something equivalent could become relevant for NLO algorithms.

⟨*Processes: process term: TBP*⟩+≡
```
    procedure :: init => process_term_init
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_term_init &
         (term, i_term_global, i_component, i_term, core, model)
      class(process_term_t), intent(inout), target :: term
      integer, intent(in) :: i_term_global
      integer, intent(in) :: i_component
      integer, intent(in) :: i_term
      class(prc_core_t), intent(in) :: core
      type(model_t), intent(in), target :: model
      type(var_list_t), pointer :: var_list
      term%i_term_global = i_term_global
      term%i_component = i_component
      term%i_term = i_term
      call core%get_constants (term%data, i_term)
      var_list => model_get_var_list_ptr (model)
      if (var_list_exists (var_list, var_str ("alphas"))) then
         term%alpha_s = var_list_get_rval (var_list, var_str ("alphas"))
      else
         term%alpha_s = -1
      end if
      call term%setup_interaction (core, model)
  !      if (term%rearrange) then
  !         call term%setup_effective_interaction (core, term%int, term%int_eff)
  !      end if
    end subroutine process_term_init
```

We fetch the process constants which determine the quantum numbers and use those to create the interaction. The interaction contains incoming and outgoing particles, no virtuals. The incoming particles are parents of the outgoing ones.

Keeping previous WHIZARD conventions, we invert the color assignment (but not flavor or helicity) for the incoming particles. When the color-flow square matrix is evaluated, this inversion is done again, so in the color-flow sequence we get the color assignments of the matrix element.

⟨*Processes: process term: TBP*⟩+≡
```
    procedure :: setup_interaction => process_term_setup_interaction
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_term_setup_interaction (term, core, model)
      class(process_term_t), intent(inout) :: term
      class(prc_core_t), intent(in) :: core
      type(model_t), intent(in), target :: model
      integer :: n_tot
      type(flavor_t), dimension(:), allocatable :: flv
      type(color_t), dimension(:), allocatable :: col
      type(helicity_t), dimension(:), allocatable :: hel
      type(quantum_numbers_t), dimension(:), allocatable :: qn
      integer :: i, n, f, h, c
      associate (data => term%data)
        n_tot = data%n_in + data%n_out
        n = 0
        do f = 1, data%n_flv
           do h = 1, data%n_hel
              do c = 1, data%n_col
```

```
                if (core%is_allowed (term%i_term, f, h, c))  n = n + 1
           end do
        end do
     end do
     allocate (term%flv (n), term%col (n), term%hel (n))
     term%n_allowed = n
     allocate (flv (n_tot), col (n_tot), hel (n_tot))
     allocate (qn (n_tot))
     call interaction_init &
          (term%int, data%n_in, 0, data%n_out, set_relations=.true.)
     i = 0
     do f = 1, data%n_flv
        do h = 1, data%n_hel
           do c = 1, data%n_col
              if (core%is_allowed (term%i_term, f, h, c)) then
                 i = i + 1
                 term%flv(i) = f
                 term%hel(i) = h
                 term%col(i) = c
                 call flavor_init (flv, data%flv_state(:,f), model)
                 call color_init_from_array (col, data%col_state(:,:,c), &
                      data%ghost_flag(:,c))
                 call color_invert (col(:data%n_in))
                 call helicity_init (hel, data%hel_state(:,h))
                 call quantum_numbers_init (qn, flv, col, hel)
                 call interaction_add_state (term%int, qn)
              end if
           end do
        end do
     end do
     call interaction_freeze (term%int)
   end associate
 end subroutine process_term_setup_interaction
```

15.5.8   Default Iterations

If the user does not specify the passes and iterations for integration, we should
be able to give reasonable defaults. These depend on the process, therefore
we implement the following procedures as methods of the process object. The
algorithm is not very sophisticated yet, it may be improved by looking at the
process in more detail.

We investigate only the first process component, assuming that it character-
izes the complexity of the process reasonable well.

The number of passes is limited to two: one for adaption, one for integration.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_n_pass_default => process_get_n_pass_default
    procedure :: adapt_grids_default => process_adapt_grids_default
    procedure :: adapt_weights_default => process_adapt_weights_default
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_pass_default (process) result (n_pass)
      class(process_t), intent(in) :: process
```

```
    integer :: n_pass
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config
    n_eff = config%get_n_tot () - 2
    select case (n_eff)
    case (1)
       n_pass = 1
    case default
       n_pass = 2
    end select
  end function process_get_n_pass_default

  function process_adapt_grids_default (process, pass) result (flag)
    class(process_t), intent(in) :: process
    integer, intent(in) :: pass
    logical :: flag
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config
    n_eff = config%get_n_tot () - 2
    select case (n_eff)
    case (1)
       flag = .false.
    case default
       select case (pass)
       case (1);  flag = .true.
       case (2);  flag = .false.
       case default
          call msg_bug ("adapt grids default: impossible pass index")
       end select
    end select
  end function process_adapt_grids_default

  function process_adapt_weights_default (process, pass) result (flag)
    class(process_t), intent(in) :: process
    integer, intent(in) :: pass
    logical :: flag
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config
    n_eff = config%get_n_tot () - 2
    select case (n_eff)
    case (1)
       flag = .false.
    case default
       select case (pass)
       case (1);  flag = .true.
       case (2);  flag = .false.
       case default
          call msg_bug ("adapt weights default: impossible pass index")
       end select
    end select
  end function process_adapt_weights_default
```

The number of iterations and calls per iteration depends on the number of outgoing particles.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_n_it_default => process_get_n_it_default
    procedure :: get_n_calls_default => process_get_n_calls_default
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_it_default (process, pass) result (n_it)
      class(process_t), intent(in) :: process
      integer, intent(in) :: pass
      integer :: n_it
      integer :: n_eff
      type(process_component_def_t), pointer :: config
      config => process%component(1)%config
      n_eff = config%get_n_tot () - 2
      select case (pass)
      case (1)
         select case (n_eff)
         case (1);   n_it = 1
         case (2);   n_it = 3
         case (3);   n_it = 5
         case (4:5); n_it = 10
         case (6);   n_it = 15
         case (7:);  n_it = 20
         end select
      case (2)
         select case (n_eff)
         case (:3);   n_it = 3
         case (4:);   n_it = 5
         end select
      end select
    end function process_get_n_it_default

    function process_get_n_calls_default (process, pass) result (n_calls)
      class(process_t), intent(in) :: process
      integer, intent(in) :: pass
      integer :: n_calls
      integer :: n_eff
      type(process_component_def_t), pointer :: config
      config => process%component(1)%config
      n_eff = config%get_n_tot () - 2
      select case (pass)
      case (1)
         select case (n_eff)
         case (1);   n_calls =   100
         case (2);   n_calls =  1000
         case (3);   n_calls =  5000
         case (4);   n_calls = 10000
         case (5);   n_calls = 20000
         case (6:);  n_calls = 50000
         end select
      case (2)
         select case (n_eff)
```

2038

```
        case (:3);  n_calls =   10000
        case (4);   n_calls =   20000
        case (5);   n_calls =   50000
        case (6);   n_calls =  100000
        case (7:);  n_calls =  200000
      end select
    end select
  end function process_get_n_calls_default
```

15.5.9   Constant process data

The following methods return basic process data that stay constant after initialization.

The process and IDs.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_id => process_get_id
    procedure :: get_num_id => process_get_num_id
    procedure :: get_run_id => process_get_run_id
    procedure :: get_library_name => process_get_library_name
```
⟨*Processes: procedures*⟩+≡
```
    function process_get_id (process) result (id)
      class(process_t), intent(in) :: process
      type(string_t) :: id
      id = process%meta%id
    end function process_get_id

    function process_get_num_id (process) result (id)
      class(process_t), intent(in) :: process
      integer :: id
      id = process%meta%num_id
    end function process_get_num_id

    function process_get_run_id (process) result (id)
      class(process_t), intent(in) :: process
      type(string_t) :: id
      id = process%meta%run_id
    end function process_get_run_id

    function process_get_library_name (process) result (id)
      class(process_t), intent(in) :: process
      type(string_t) :: id
      id = process%meta%lib%get_name ()
    end function process_get_library_name
```

The number of incoming particles.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_n_in => process_get_n_in
```
⟨*Processes: procedures*⟩+≡
```
    function process_get_n_in (process) result (n)
      class(process_t), intent(in) :: process
```

```
      integer :: n
      n = process%config%n_in
    end function process_get_n_in
```

The number of MCI data sets.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_n_mci => process_get_n_mci
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_mci (process) result (n)
      class(process_t), intent(in) :: process
      integer :: n
      n = process%config%n_mci
    end function process_get_n_mci
```

The number of process components, total.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_n_components => process_get_n_components
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_components (process) result (n)
      class(process_t), intent(in) :: process
      integer :: n
      n = process%meta%n_components
    end function process_get_n_components
```

The number of process terms, total.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_n_terms => process_get_n_terms
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_n_terms (process) result (n)
      class(process_t), intent(in) :: process
      integer :: n
      n = process%config%n_terms
    end function process_get_n_terms
```

Return the indices of the components that belong to a specific MCI entry.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_i_component => process_get_i_component
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_get_i_component (process, i_mci, i_component)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      integer, dimension(:), intent(out), allocatable :: i_component
      associate (mci_entry => process%mci_entry(i_mci))
        allocate (i_component (size (mci_entry%i_component)))
        i_component = mci_entry%i_component
      end associate
    end subroutine process_get_i_component
```

Return the ID of a specific component.

⟨Processes: process: TBP⟩+≡
```
    procedure :: get_component_id => process_get_component_id
```

⟨Processes: procedures⟩+≡
```
    function process_get_component_id (process, i_component) result (id)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_component
      type(string_t) :: id
      id = process%meta%component_id(i_component)
    end function process_get_component_id
```

Return a pointer to the definition of a specific component.

⟨Processes: process: TBP⟩+≡
```
    procedure :: get_component_def_ptr => process_get_component_def_ptr
```

⟨Processes: procedures⟩+≡
```
    function process_get_component_def_ptr (process, i_component) result (ptr)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_component
      type(process_component_def_t), pointer :: ptr
      ptr => process%meta%lib%get_component_def_ptr (process%meta%id, i_component)
    end function process_get_component_def_ptr
```

These procedures extract and restore (by transferring the allocation) the process
core. This is useful for changing process parameters from outside this module.

⟨Processes: process: TBP⟩+≡
```
    procedure :: extract_component_core => process_extract_component_core
    procedure :: restore_component_core => process_restore_component_core
```

⟨Processes: procedures⟩+≡
```
    subroutine process_extract_component_core (process, i_component, core)
      class(process_t), intent(inout) :: process
      integer, intent(in) :: i_component
      class(prc_core_t), intent(inout), allocatable :: core
      call move_alloc (from = process%component(i_component)%core, to = core)
    end subroutine process_extract_component_core

    subroutine process_restore_component_core (process, i_component, core)
      class(process_t), intent(inout) :: process
      integer, intent(in) :: i_component
      class(prc_core_t), intent(inout), allocatable :: core
      call move_alloc (from = core, to = process%component(i_component)%core)
    end subroutine process_restore_component_core
```

The block of process constants.

⟨Processes: process: TBP⟩+≡
```
    procedure :: get_constants => process_get_constants
```

⟨Processes: procedures⟩+≡
```
    function process_get_constants (process, i) result (data)
      class(process_t), intent(in) :: process
      integer, intent(in) :: i
      type(process_constants_t) :: data
```

2041

```
      data = process%component(i)%core%data
   end function process_get_constants
```

Return the set of outgoing flavors that are associated with a particular term.
We deduce this from the effective interaction.

⟨*Processes: process: TBP*⟩+≡
```
   procedure :: get_term_flv_out => process_get_term_flv_out
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_get_term_flv_out (process, i_term, flv)
     class(process_t), intent(in), target :: process
     integer, intent(in) :: i_term
     type(flavor_t), dimension(:,:), allocatable, intent(out) :: flv
     type(interaction_t), pointer :: int
     int => process%term(i_term)%int_eff
     if (.not. associated (int))  int => process%term(i_term)%int
     call interaction_get_flv_out (int, flv)
   end subroutine process_get_term_flv_out
```

Return true if there is any unstable particle in any of the process terms. We
decide this based on the provided model instance, not the one that is stored in
the process object.

⟨*Processes: process: TBP*⟩+≡
```
   procedure :: contains_unstable => process_contains_unstable
```

⟨*Processes: procedures*⟩+≡
```
   function process_contains_unstable (process, model) result (flag)
     class(process_t), intent(in) :: process
     type(model_t), intent(in), target :: model
     logical :: flag
     integer :: i_term
     type(flavor_t), dimension(:,:), allocatable :: flv
     flag = .false.
     do i_term = 1, process%get_n_terms ()
        call process%get_term_flv_out (i_term, flv)
        call flavor_set_model (flv, model)
        flag = .not. all (flavor_is_stable (flv))
        deallocate (flv)
        if (flag)  return
     end do
   end function process_contains_unstable
```

The nominal process energy.

⟨*Processes: process: TBP*⟩+≡
```
   procedure :: get_sqrts => process_get_sqrts
```

⟨*Processes: procedures*⟩+≡
```
   function process_get_sqrts (process) result (sqrts)
     class(process_t), intent(in) :: process
     real(default) :: sqrts
     sqrts = beam_data_get_sqrts (process%beam_config%data)
   end function process_get_sqrts
```

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: has_matrix_element => process_has_matrix_element
```

⟨*Processes: procedures*⟩+≡
```
    function process_has_matrix_element (process, i) result (flag)
      class(process_t), intent(in) :: process
      integer, intent(in), optional :: i
      logical :: flag
      if (present (i)) then
         flag = process%component(i)%active
      else
         flag = any (process%component%active)
      end if
    end function process_has_matrix_element
```

Pointer to the beam data object.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_beam_data_ptr => process_get_beam_data_ptr
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_beam_data_ptr (process) result (beam_data)
      class(process_t), intent(in), target :: process
      type(beam_data_t), pointer :: beam_data
      beam_data => process%beam_config%data
    end function process_get_beam_data_ptr
```

Return true if lab and c.m. frame coincide for this process.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: cm_frame => process_cm_frame
```

⟨*Processes: procedures*⟩+≡
```
    function process_cm_frame (process) result (flag)
      class(process_t), intent(in), target :: process
      logical :: flag
      type(beam_data_t), pointer :: beam_data
      beam_data => process%beam_config%data
      flag = beam_data_cm_frame (beam_data)
    end function process_cm_frame
```

Get the PDF set currently in use, if any.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_pdf_set => process_get_pdf_set
```

⟨*Processes: procedures*⟩+≡
```
    function process_get_pdf_set (process) result (pdf_set)
      class(process_t), intent(in) :: process
      integer :: pdf_set
      pdf_set = process%beam_config%get_pdf_set ()
    end function process_get_pdf_set
```

Pointer to the process variable list.

⟨*Processes: process: TBP*⟩+≡
```
    procedure :: get_var_list_ptr => process_get_var_list_ptr
```

⟨*Processes: procedures*⟩+≡
```
function process_get_var_list_ptr (process) result (ptr)
  class(process_t), intent(in), target :: process
  type(var_list_t), pointer :: ptr
  ptr => process%meta%var_list
end function process_get_var_list_ptr
```

Pointer to the common model.

⟨*Processes: process: TBP*⟩+≡
```
procedure :: get_model_ptr => process_get_model_ptr
```

⟨*Processes: procedures*⟩+≡
```
function process_get_model_ptr (process) result (ptr)
  class(process_t), intent(in) :: process
  type(model_t), pointer :: ptr
  ptr => process%config%model
end function process_get_model_ptr
```

Use the embedded RNG factory to spawn a new random-number generator instance. (This modifies the state of the factory.)

⟨*Processes: process: TBP*⟩+≡
```
procedure :: make_rng => process_make_rng
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_make_rng (process, rng)
  class(process_t), intent(inout) :: process
  class(rng_t), intent(out), allocatable :: rng
  if (allocated (process%config%rng_factory)) then
     call process%config%rng_factory%make (rng)
  else
     call msg_bug ("Process: make rng: factory not allocated")
  end if
end subroutine process_make_rng
```

15.5.10 Compute an amplitude

Each process variant should allow for computing an amplitude value directly, without generating a process instance.

The process component is selected by the index `i`. The term within the process component is selected by `j`. The momentum combination is transferred as the array `p`. The function sets the specific quantum state via the indices of a flavor `f`, helicity `h`, and color `c` combination. Each index refers to the list of flavor, helicity, and color states, respectively, as stored in the process data.

Optionally, we may set factorization and renormalization scale. If unset, the partonic c.m. energy is inserted.

The function checks arguments for validity. For invalid arguments (quantum states), we return zero.

⟨*Processes: process: TBP*⟩+≡
```
procedure :: compute_amplitude => process_compute_amplitude
```

⟨*Processes: procedures*⟩+≡

```
function process_compute_amplitude &
    (process, i, j, p, f, h, c, fac_scale, ren_scale) result (amp)
  class(process_t), intent(in) :: process
  integer, intent(in) :: i, j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in), optional :: fac_scale, ren_scale
  real(default) :: fscale, rscale
  complex(default) :: amp
  amp = 0
  if (0 < i .and. i <= process%meta%n_components) then
     if (process%component(i)%active) then
        associate (data => process%component(i)%core%data)
          if (size (p) == data%n_in + data%n_out &
               .and. 0 < f .and. f <= data%n_flv &
               .and. 0 < h .and. h <= data%n_hel &
               .and. 0 < c .and. c <= data%n_col) then
             if (present (fac_scale)) then
                fscale = fac_scale
             else
                fscale = sum (p(data%n_in+1:)) ** 1
             end if
             if (present (ren_scale)) then
                rscale = ren_scale
             else
                rscale = fscale
             end if
             amp = process%component(i)%core% &
                  compute_amplitude (j, p, f, h, c, fscale, rscale)
          end if
        end associate
     else
        amp = 0
     end if
  end if
end function process_compute_amplitude
```

This is for suppression of numerical noise in the integration results stored in the `process_mci_entry` type. As the error and efficiency enter the MD5 sum, we recompute it.

⟨*Processes: process: TBP*⟩+≡

```
procedure :: pacify => process_pacify
```

⟨*Processes: procedures*⟩+≡

```
subroutine process_pacify (process, efficiency_reset, error_reset)
  class(process_t), intent(inout) :: process
  logical, intent(in), optional :: efficiency_reset, error_reset
  logical :: eff_reset, err_reset
  integer :: i
  eff_reset = .false.
  err_reset = .false.
  if (present (efficiency_reset))  eff_reset = efficiency_reset
  if (present (error_reset))  err_reset = error_reset
```

```
   if (allocated (process%mci_entry)) then
      do i = 1, size (process%mci_entry)
         call process%mci_entry(i)%results%pacify (efficiency_reset)
         if (allocated (process%mci_entry(i)%mci)) then
            if (process%mci_entry(i)%mci%error_known .and. err_reset) &
               process%mci_entry(i)%mci%error = 0
            if (process%mci_entry(i)%mci%efficiency_known .and. &
               eff_reset)  process%mci_entry(i)%mci%efficiency = 1
            select type (mci => process%mci_entry(i)%mci)
            type is (mci_vamp_t)
               call mci%pacify (efficiency_reset, error_reset)
               call mci%compute_md5sum ()
            end select
         end if
      end do
   end if
 end subroutine process_pacify
```

15.5.11   Process instances

Kinematics instance

In this data type we combine all objects (instances) necessary for generating (or recovering) a kinematical configuration. The components work together as an implementation of multi-channel phase space.

sf_chain is an instance of the structure-function chain. It is used both for generating kinematics and, after the proper scale has been determined, evaluating the structure function entries.

phs is an instance of the phase space for the elementary process.

The array f contains the products of the Jacobians that originate from parameter mappings in the structure-function chain or in the phase space. We allocate this explicitly if either sf_chain or phs are explicitly allocated, otherwise we can take over a pointer.

All components are implemented as pointers to (anonymous) targets. For each component, there is a flag that tells whether this component is to be regarded as a proper component ('owned' by the object) or as a pointer.

⟨*Processes: types*⟩+≡
```
   type :: kinematics_t
      integer :: n_in = 0
      integer :: n_channel = 0
      integer :: selected_channel = 0
      type(sf_chain_instance_t), pointer :: sf_chain => null ()
      class(phs_t), pointer :: phs => null ()
      real(default), dimension(:), pointer :: f => null ()
      real(default) :: phs_factor
      logical :: sf_chain_allocated = .false.
      logical :: phs_allocated = .false.
      logical :: f_allocated = .false.
    contains
    ⟨Processes: kinematics: TBP⟩
   end type kinematics_t
```

Output. Show only those components which are marked as owned.

⟨*Processes: kinematics: TBP*⟩≡
```
procedure :: write => kinematics_write
```

⟨*Processes: procedures*⟩+≡
```
subroutine kinematics_write (object, unit)
  class(kinematics_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, c
  u = output_unit (unit)
  if (object%f_allocated) then
      write (u, "(1x,A)")  "Flux * PHS volume:"
      write (u, "(2x,ES19.12)")  object%phs_factor
      write (u, "(1x,A)")  "Jacobian factors per channel:"
      do c = 1, size (object%f)
          write (u, "(3x,I0,':',1x,ES13.7)", advance="no")  c, object%f(c)
          if (c == object%selected_channel) then
             write (u, "(1x,A)")  "[selected]"
          else
             write (u, *)
          end if
      end do
  end if
  if (object%sf_chain_allocated) then
      call write_separator (u)
      call object%sf_chain%write (u)
  end if
  if (object%phs_allocated) then
      call write_separator (u)
      call object%phs%write (u)
  end if
end subroutine kinematics_write
```

Finalizer. Delete only those components which are marked as owned.

⟨*Processes: kinematics: TBP*⟩+≡
```
procedure :: final => kinematics_final
```

⟨*Processes: procedures*⟩+≡
```
subroutine kinematics_final (object)
  class(kinematics_t), intent(inout) :: object
  if (object%sf_chain_allocated) then
      call object%sf_chain%final ()
      deallocate (object%sf_chain)
      object%sf_chain_allocated = .false.
  end if
  if (object%phs_allocated) then
      call object%phs%final ()
      deallocate (object%phs)
      object%phs_allocated = .false.
  end if
  if (object%f_allocated) then
      deallocate (object%f)
      object%f_allocated = .false.
```

2047

```
      end if
    end subroutine kinematics_final
```

Allocate the structure-function chain instance, initialize it as a copy of the `sf_chain` template, and prepare it for evaluation.

The `sf_chain` remains a target because the (usually constant) beam momenta are taken from there.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: init_sf_chain => kinematics_init_sf_chain
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_init_sf_chain (k, core, sf_chain, config, tmp)
      class(kinematics_t), intent(inout) :: k
      class(prc_core_t), intent(in) :: core
      type(sf_chain_t), intent(in), target :: sf_chain
      type(process_beam_config_t), intent(in) :: config
      class(workspace_t), intent(inout), allocatable :: tmp
      integer :: n_strfun, n_channel
      integer :: c
      k%n_in = beam_data_get_n_in (config%data)
      n_strfun = config%n_strfun
      n_channel = config%n_channel
      allocate (k%sf_chain)
      k%sf_chain_allocated = .true.
      call core%init_sf_chain (k%sf_chain, sf_chain, n_channel, tmp)
      if (n_strfun /= 0) then
         do c = 1, n_channel
            call k%sf_chain%set_channel (c, config%sf_channel(c))
         end do
      end if
      call k%sf_chain%link_interactions ()
      call k%sf_chain%exchange_mask ()
      call k%sf_chain%init_evaluators ()
    end subroutine kinematics_init_sf_chain
```

Allocate and initialize the phase-space part and the array of Jacobian factors.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: init_phs => kinematics_init_phs
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_init_phs (k, config)
      class(kinematics_t), intent(inout) :: k
      class(phs_config_t), intent(in), target :: config
      k%n_channel = config%get_n_channel ()
      call config%allocate_instance (k%phs)
      call k%phs%init (config)
      k%phs_allocated = .true.
      allocate (k%f (k%n_channel))
      k%f = 0
      k%f_allocated = .true.
    end subroutine kinematics_init_phs
```

Initialize the kinematics in form of simple pointers. In essence, this is a shallow copy, but we have to set the flags correctly to indicate this fact.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: init_ptr => kinematics_init_ptr
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_init_ptr (k, k_in)
      class(kinematics_t), intent(out) :: k
      type(kinematics_t), intent(in) :: k_in
      k%n_in = k_in%n_in
      k%n_channel = k_in%n_channel
      k%sf_chain => k_in%sf_chain
      k%phs => k_in%phs
      k%f => k_in%f
    end subroutine kinematics_init_ptr
```

Generate kinematics, given a phase-space channel and a MC parameter set. The main result is the momentum array `p`, but we also fill the momentum entries in the structure-function chain and the Jacobian-factor array `f`. Regarding phase space, We fill only the parameter arrays for the selected channel.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: compute_selected_channel => kinematics_compute_selected_channel
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_compute_selected_channel &
        (k, mci_work, phs_channel, p, success)
      class(kinematics_t), intent(inout) :: k
      type(mci_work_t), intent(in) :: mci_work
      integer, intent(in) :: phs_channel
      type(vector4_t), dimension(:), intent(out) :: p
      logical, intent(out) :: success
      integer :: sf_channel
      k%selected_channel = phs_channel
      sf_channel = k%phs%config%get_sf_channel (phs_channel)
      call k%sf_chain%compute_kinematics (sf_channel, mci_work%get_x_strfun ())
      call k%sf_chain%get_out_momenta (p(1:k%n_in))
      call k%phs%set_incoming_momenta (p(1:k%n_in))
      call k%phs%compute_flux ()
      call k%phs%select_channel (phs_channel)
      call k%phs%evaluate_selected_channel &
          (phs_channel, mci_work%get_x_process ())
      if (k%phs%q_defined) then
         call k%phs%get_outgoing_momenta (p(k%n_in+1:))
         k%phs_factor = k%phs%get_overall_factor ()
         success = .true.
      else
         k%phs_factor = 0
         success = .false.
      end if
    end subroutine kinematics_compute_selected_channel
```

Complete kinematics by filling the non-selected phase-space parameter arrays.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: compute_other_channels => kinematics_compute_other_channels
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_compute_other_channels (k, mci_work, phs_channel)
      class(kinematics_t), intent(inout) :: k
      type(mci_work_t), intent(in) :: mci_work
      integer, intent(in) :: phs_channel
      integer :: c, c_sf
      call k%phs%evaluate_other_channels (phs_channel)
      do c = 1, k%n_channel
         c_sf = k%phs%config%get_sf_channel (c)
         k%f(c) = k%sf_chain%get_f (c_sf) * k%phs%get_f (c)
      end do
    end subroutine kinematics_compute_other_channels
```

Just fetch the outgoing momenta of the `sf_chain` subobject, which become the incoming (seed) momenta of the hard interaction.

This is a stripped down-version of the above which we use when recovering kinematics. Momenta are known, but no MC parameters yet.

(We do not use the `get_out_momenta` method of the chain, since this relies on the structure-function interactions, which are not necessary filled here. We do rely on the momenta of the last evaluator in the chain, however.)

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: get_incoming_momenta => kinematics_get_incoming_momenta
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_get_incoming_momenta (k, p)
      class(kinematics_t), intent(in) :: k
      type(vector4_t), dimension(:), intent(out) :: p
      type(interaction_t), pointer :: int
      integer :: i
      int => k%sf_chain%get_out_int_ptr ()
      do i = 1, k%n_in
         p(i) = interaction_get_momentum (int, k%sf_chain%get_out_i (i))
      end do
    end subroutine kinematics_get_incoming_momenta
```

This inverts the remainder of the above `compute` method. We know the momenta and recover the rest, as far as needed. If we select a channel, we can complete the inversion and reconstruct the MC parameter set.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: recover_mcpar => kinematics_recover_mcpar
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_recover_mcpar (k, mci_work, phs_channel, p)
      class(kinematics_t), intent(inout) :: k
      type(mci_work_t), intent(inout) :: mci_work
      integer, intent(in) :: phs_channel
      type(vector4_t), dimension(:), intent(in) :: p
      integer :: c, c_sf
      real(default), dimension(:), allocatable :: x_sf, x_phs
      c = phs_channel
      c_sf = k%phs%config%get_sf_channel (c)
      k%selected_channel = c
      call k%sf_chain%recover_kinematics (c_sf)
```

```
      call k%phs%set_incoming_momenta (p(1:k%n_in))
      call k%phs%compute_flux ()
      call k%phs%set_outgoing_momenta (p(k%n_in+1:))
      call k%phs%inverse ()
      do c = 1, k%n_channel
         c_sf = k%phs%config%get_sf_channel (c)
         k%f(c) = k%sf_chain%get_f (c_sf) * k%phs%get_f (c)
      end do
      k%phs_factor = k%phs%get_overall_factor ()
      c = phs_channel
      c_sf = k%phs%config%get_sf_channel (c)
      allocate (x_sf (k%sf_chain%config%get_n_bound ()))
      allocate (x_phs (k%phs%config%get_n_par ()))
      call k%phs%select_channel (c)
      call k%sf_chain%get_mcpar (c_sf, x_sf)
      call k%phs%get_mcpar (c, x_phs)
      call mci_work%set_x_strfun (x_sf)
      call mci_work%set_x_process (x_phs)
    end subroutine kinematics_recover_mcpar
```

Retrieve the MC input parameter array for a specific channel. We assume that the kinematics is complete, so this is known for all channels.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: get_mcpar => kinematics_get_mcpar
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_get_mcpar (k, phs_channel, r)
      class(kinematics_t), intent(in) :: k
      integer, intent(in) :: phs_channel
      real(default), dimension(:), intent(out) :: r
      integer :: sf_channel, n_par_sf, n_par_phs
      sf_channel = k%phs%config%get_sf_channel (phs_channel)
      n_par_phs = k%phs%config%get_n_par ()
      n_par_sf = k%sf_chain%config%get_n_bound ()
      if (n_par_sf > 0) then
         call k%sf_chain%get_mcpar (sf_channel, r(1:n_par_sf))
      end if
      if (n_par_phs > 0) then
         call k%phs%get_mcpar (phs_channel, r(n_par_sf+1:))
      end if
    end subroutine kinematics_get_mcpar
```

Evaluate the structure function chain, assuming that kinematics is known.

The status must be precisely `SF_DONE_KINEMATICS`. We thus avoid evaluating the chain twice via different pointers to the same target.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: evaluate_sf_chain => kinematics_evaluate_sf_chain
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_evaluate_sf_chain (k, fac_scale)
      class(kinematics_t), intent(inout) :: k
      real(default), intent(in) :: fac_scale
      select case (k%sf_chain%get_status ())
      case (SF_DONE_KINEMATICS)
```

```
        call k%sf_chain%evaluate (fac_scale)
      end select
    end subroutine kinematics_evaluate_sf_chain
```

Recover beam momenta, i.e., return the beam momenta stored in the current `sf_chain` to their source. This is a side effect.

⟨*Processes: kinematics: TBP*⟩+≡
```
    procedure :: return_beam_momenta => kinematics_return_beam_momenta
```

⟨*Processes: procedures*⟩+≡
```
    subroutine kinematics_return_beam_momenta (k)
      class(kinematics_t), intent(in) :: k
      call k%sf_chain%return_beam_momenta ()
    end subroutine kinematics_return_beam_momenta
```

Process component instance

The actual calculation of a sampling point is done from here.

The `config` pointer accesses the corresponding configuration in the `process` object.

The `active` flag indicates that we are currently computing this component, together with all other components that share the same MC parameter set. Inactive components are using a different MC parameter set and are not in use for this sampling point.

The `k_seed` subobject contains the kinematics (structure-function chain, phase space, etc.) that implements the 'seed' configuration of momenta. This version of the process kinematics is accessed by the MCI setup.

`p_seed` is the array of momenta that we compute from the MC input parameters, via the `k_seed` subobject. Depending on the process variant, these may or may not coincide with the momenta that enter the process terms associated to this component.

The `tmp` object can be used for storing intermediate results. Its precise type and contents depend on the process variant.

⟨*Processes: types*⟩+≡
```
    type :: component_instance_t
      type(process_component_t), pointer :: config => null ()
      logical :: active = .false.
      type(kinematics_t) :: k_seed
      type(vector4_t), dimension(:), allocatable :: p_seed
      logical :: sqme_known = .false.
      real(default) :: sqme = 0
      class(workspace_t), allocatable :: tmp
    contains
      ⟨Processes: component instance: TBP⟩
    end type component_instance_t
```

In the header, fetch the component index from the configuration record. `process_component_t` configuration block.

We write the `sf_chain` subobject only upon request, since its instances appear elsewhere.

⟨*Processes: component instance: TBP*⟩≡

```
          procedure :: write => component_instance_write
⟨Processes: procedures⟩+≡
      subroutine component_instance_write (object, unit, testflag)
        class(component_instance_t), intent(in) :: object
        integer, intent(in), optional :: unit
        logical, intent(in), optional :: testflag
        integer :: u, i
        u = output_unit (unit)
        if (object%active) then
           if (associated (object%config)) then
              write (u, "(1x,A,I0)")  "Component #", object%config%index
           else
              write (u, "(1x,A)")  "Component [undefined]"
           end if
        else
           write (u, "(1x,A,I0,A)")  "Component #", object%config%index, &
                " [inactive]"
        end if
        if (allocated (object%p_seed)) then
           write (u, "(1x,A)")  "Seed momenta:"
           do i = 1, size (object%p_seed)
              call vector4_write (object%p_seed(i), u, testflag = testflag)
           end do
        end if
        write (u, "(1x,A)")  "Squared matrix element:"
        if (object%sqme_known) then
           write (u, "(2x,ES19.12)")  object%sqme
        else
           write (u, "(2x,A)")  "[undefined]"
        end if
        call object%k_seed%write (u)
        if (allocated (object%tmp)) then
           call write_separator (u)
           call object%tmp%write (u)
        end if
      end subroutine component_instance_write
```

Finalizer

⟨*Processes: component instance: TBP*⟩+≡
```
      procedure :: final => component_instance_final
```

⟨*Processes: procedures*⟩+≡
```
      subroutine component_instance_final (object)
        class(component_instance_t), intent(inout) :: object
        call object%k_seed%final ()
      end subroutine component_instance_final
```

Initialize: associate the configuration pointer. Also initialize the process workspace, if there is anything to do. This initialization is a matter of the process core.

⟨*Processes: component instance: TBP*⟩+≡
```
      procedure :: init => component_instance_init
```

⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_init (component, config)
      class(component_instance_t), intent(out) :: component
      type(process_component_t), intent(in), target :: config
      integer :: n_in, n_tot
      component%config => config
      associate (core => component%config%core)
        n_in = core%data%n_in
        n_tot = n_in + core%data%n_out
        allocate (component%p_seed (n_tot))
        call core%allocate_workspace (component%tmp)
      end associate
    end subroutine component_instance_init
```

Initialize the seed-kinematics configuration. All subobjects are allocated explicitly.

⟨*Processes: component instance: TBP*⟩+≡
```
    procedure :: setup_kinematics => component_instance_setup_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_setup_kinematics (component, sf_chain, config)
      class(component_instance_t), intent(inout) :: component
      type(sf_chain_t), intent(in), target :: sf_chain
      type(process_beam_config_t), intent(in) :: config
      call component%k_seed%init_sf_chain &
           (component%config%core, sf_chain, config, component%tmp)
      call component%k_seed%init_phs (component%config%phs_config)
    end subroutine component_instance_setup_kinematics
```

Setup seed kinematics, starting from the MC parameter set given as argument. As a result, the `k_seed` kinematics object is evaluated (except for the structure-function matrix-element evaluation, which we postpone until we know the factorization scale), and we have a valid `p_seed` momentum array.

⟨*Processes: component instance: TBP*⟩+≡
```
    procedure :: compute_seed_kinematics => &
         component_instance_compute_seed_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_compute_seed_kinematics &
         (component, mci_work, phs_channel, success)
      class(component_instance_t), intent(inout), target :: component
      type(mci_work_t), intent(in) :: mci_work
      integer, intent(in) :: phs_channel
      logical, intent(out) :: success
      call component%k_seed%compute_selected_channel &
           (mci_work, phs_channel, component%p_seed, success)
    end subroutine component_instance_compute_seed_kinematics
```

Inverse: recover missing parts of the kinematics, given a complete set of seed momenta. Select a channel and reconstruct the MC parameter set.

⟨*Processes: component instance: TBP*⟩+≡
```
    procedure :: recover_mcpar => component_instance_recover_mcpar
```

2054

⟨*Processes: procedures*⟩+≡
```
subroutine component_instance_recover_mcpar (component, mci_work, phs_channel)
  class(component_instance_t), intent(inout), target :: component
  type(mci_work_t), intent(inout) :: mci_work
  integer, intent(in) :: phs_channel
  call component%k_seed%recover_mcpar &
       (mci_work, phs_channel, component%p_seed)
end subroutine component_instance_recover_mcpar
```

Compute the momenta in the hard interactions, one for each term that constitutes this process component. In simple cases this amounts to just copying momenta. In more advanced cases, we may generate distinct sets of momenta from the seed kinematics.

The interactions in the term instances are accessed individually. We may choose to calculate all terms at once together with the seed kinematics, use `component%tmp` for storage, and just fill the interactions here.

⟨*Processes: component instance: TBP*⟩+≡
```
procedure :: compute_hard_kinematics => &
     component_instance_compute_hard_kinematics
```

⟨*Processes: procedures*⟩+≡
```
subroutine component_instance_compute_hard_kinematics &
     (component, term, skip_term)
  class(component_instance_t), intent(inout) :: component
  type(term_instance_t), dimension(:), intent(inout) :: term
  integer, intent(in), optional :: skip_term
  integer :: j, i
  associate (core => component%config%core)
    associate (i_term => component%config%i_term)
      do j = 1, size (i_term)
         i = i_term(j)
         if (present (skip_term)) then
            if (i == skip_term)  cycle
         end if
         call core%compute_hard_kinematics &
              (component%p_seed, i, term(i)%int_hard, component%tmp)
      end do
    end associate
  end associate
end subroutine component_instance_compute_hard_kinematics
```

Here, we invert this. We fetch the incoming momenta which reside in the appropriate `sf_chain` object, stored within the `k_seed` subobject. On the other hand, we have the outgoing momenta of the effective interaction. We rely on the process core to compute the remaining seed momenta and to fill the momenta within the hard interaction. (The latter is trivial if hard and effective interaction coincide.)

After this is done, the incoming momenta in the trace evaluator that corresponds to the hard (effective) interaction, are still left undefined. We remedy this by calling `receive_kinematics` once.

⟨*Processes: component instance: TBP*⟩+≡
```
procedure :: recover_seed_kinematics => &
```

```
            component_instance_recover_seed_kinematics
```
⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_recover_seed_kinematics (component, term)
      class(component_instance_t), intent(inout) :: component
      type(term_instance_t), intent(inout) :: term
      integer :: n_in
      n_in = component%k_seed%n_in
      call component%k_seed%get_incoming_momenta (component%p_seed(1:n_in))
      associate (core => component%config%core)
        call core%recover_kinematics &
             (component%p_seed, term%int_hard, term%isolated%int_eff, &
             component%tmp)
        call term%isolated%receive_kinematics ()
      end associate
    end subroutine component_instance_recover_seed_kinematics
```

Compute the integration parameters for all channels except the selected one.

⟨*Processes: component instance: TBP*⟩+≡
```
    procedure :: compute_other_channels => &
         component_instance_compute_other_channels
```
⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_compute_other_channels &
         (component, mci_work, phs_channel)
      class(component_instance_t), intent(inout), target :: component
      type(mci_work_t), intent(in) :: mci_work
      integer, intent(in) :: phs_channel
      call component%k_seed%compute_other_channels (mci_work, phs_channel)
    end subroutine component_instance_compute_other_channels
```

Recover beam momenta, i.e., return the beam momenta as currently stored in
the kinematics subobject to their source. This is a side effect.

⟨*Processes: component instance: TBP*⟩+≡
```
    procedure :: return_beam_momenta => component_instance_return_beam_momenta
```
⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_return_beam_momenta (component)
      class(component_instance_t), intent(in) :: component
      call component%k_seed%return_beam_momenta ()
    end subroutine component_instance_return_beam_momenta
```

Evaluate the trace of the transition matrix, convoluted with the initial state,
and summed over all terms. The trace evaluators of the individual terms have
only a single matrix element. We implicitly drop the imaginary part of the
terms, which should be zero anyway.

⟨*Processes: component instance: TBP*⟩+≡
```
    procedure :: evaluate_sqme => component_instance_evaluate_sqme
```
⟨*Processes: procedures*⟩+≡
```
    subroutine component_instance_evaluate_sqme (component, term)
      class(component_instance_t), intent(inout) :: component
      type(term_instance_t), dimension(:), intent(in), target :: term
      type(interaction_t), pointer :: int
```

```
      real(default) :: sqme
      integer :: j, i
      component%sqme = 0
      associate (i_term => component%config%i_term)
        do j = 1, size (i_term)
           i = i_term(j)
           if (term(i)%passed) then
              int => evaluator_get_int_ptr (term(i)%connected%trace)
              sqme = interaction_get_matrix_element (int, 1)
              component%sqme = component%sqme + sqme * term(i)%weight
           end if
        end do
      end associate
      component%sqme_known = .true.
    end subroutine component_instance_evaluate_sqme
```

Term instance

A `term_instance_t` object contains all data that describe a term. Each process component consists of one or more distinct terms which may differ in kinematics, but whose squared transition matrices have to be added pointwise.

The `active` flag is set when this term is connected to an active process component. Inactive terms are skipped for kinematics and evaluation.

The `k_term` object is the instance of the kinematics setup (structure-function chain, phase space, etc.) that applies specifically to this term. In ordinary cases, it consists of straight pointers to the seed kinematics.

The `amp` array stores the amplitude values when we get them from evaluating the associated matrix-element code.

The `int_hard` interaction describes the elementary hard process. It receives the momenta and the amplitude entries for each sampling point.

The `isolated` object holds the effective parton state for the elementary interaction. The amplitude entries are computed from `int_hard`.

The `connected` evaluator set convolutes this scattering matrix with the beam (and possibly structure-function) density matrix.

The `checked` flag is set once we have applied cuts on this term. The result of this is stored in the `passed` flag. Once the term has passed cuts, we calculate the various scale and weight expressions.

⟨*Processes: types*⟩+≡
```
    type :: term_instance_t
      type(process_term_t), pointer :: config => null ()
      logical :: active = .false.
      type(kinematics_t) :: k_term
      complex(default), dimension(:), allocatable :: amp
      type(interaction_t) :: int_hard
      type(isolated_state_t) :: isolated
      type(connected_state_t) :: connected
      logical :: checked = .false.
      logical :: passed = .false.
      real(default) :: scale = 0
      real(default) :: fac_scale = 0
      real(default) :: ren_scale = 0
```

2057

```
      real(default) :: weight = 1
      type(vector4_t), dimension(:), allocatable :: p_hard
   contains
   ⟨Processes: term instance: TBP⟩
   end type term_instance_t
```

⟨*Processes: term instance: TBP*⟩≡
```
   procedure :: write => term_instance_write
```

⟨*Processes: procedures*⟩+≡
```
   subroutine term_instance_write (term, unit, show_eff_state, testflag)
     class(term_instance_t), intent(in) :: term
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: show_eff_state
     logical, intent(in), optional :: testflag
     integer :: u
     logical :: state
     u = output_unit (unit)
     state = .true.;  if (present (show_eff_state))  state = show_eff_state
     if (term%active) then
        if (associated (term%config)) then
           write (u, "(1x,A,I0,A,I0,A)")  "Term #", term%config%i_term, &
                " (component #", term%config%i_component, ")"
        else
           write (u, "(1x,A)")  "Term [undefined]"
        end if
     else
        write (u, "(1x,A,I0,A)")  "Term #", term%config%i_term, &
             " [inactive]"
     end if
     if (term%checked) then
        write (u, "(3x,A,L1)")      "passed cuts          = ", term%passed
     end if
     if (term%passed) then
        write (u, "(3x,A,ES19.12)")  "overall scale        = ", term%scale
        write (u, "(3x,A,ES19.12)")  "factorization scale  = ", term%fac_scale
        write (u, "(3x,A,ES19.12)")  "renormalization scale = ", term%ren_scale
        write (u, "(3x,A,ES19.12)")  "reweighting factor   = ", term%weight
     end if
     call term%k_term%write (u)
     call write_separator (u)
     write (u, "(1x,A)")  "Amplitude (transition matrix of the &
          &hard interaction):"
     call write_separator (u)
     call interaction_write (term%int_hard, u, testflag = testflag)
     if (state .and. term%isolated%has_trace) then
        call write_separator (u)
        write (u, "(1x,A)")  "Evaluators for the hard interaction:"
        call term%isolated%write (u, testflag = testflag)
     end if
     if (state .and. term%connected%has_trace) then
        call write_separator (u)
        write (u, "(1x,A)")  "Evaluators for the connected process:"
        call term%connected%write (u, testflag = testflag)
```

2058

```
      end if
   end subroutine term_instance_write
```

The interactions and evaluators must be finalized.

⟨*Processes: term instance: TBP*⟩+≡
```
   procedure :: final => term_instance_final
```

⟨*Processes: procedures*⟩+≡
```
   subroutine term_instance_final (term)
     class(term_instance_t), intent(inout) :: term
     call term%k_term%final ()
     call term%connected%final ()
     call term%isolated%final ()
     call interaction_final (term%int_hard)
   end subroutine term_instance_final
```

For initialization, we make use of defined assignment for the `interaction_t` type. This creates a deep copy.

The hard interaction (incoming momenta) is linked to the structure function instance. In the isolated state, we either set pointers to both, or we create modified copies (`rearrange`) as effective structure-function chain and interaction, respectively.

Finally, we set up the `subevt` component that will be used for evaluating observables, collecting particles from the trace evaluator in the effective connected state. Their quantum numbers must be determined by following back source links and set explicitly, since they are already eliminated in that trace.

The `rearrange` parts are still commented out; they could become relevant for a NLO algorithm.

⟨*Processes: term instance: TBP*⟩+≡
```
   procedure :: init => term_instance_init
```

⟨*Processes: procedures*⟩+≡
```
   subroutine term_instance_init (term, &
        config, k_seed, beam_config, core, process_var_list)
     class(term_instance_t), intent(out), target :: term
     type(process_term_t), intent(in), target :: config
     type(kinematics_t), intent(in) :: k_seed
     type(process_beam_config_t), intent(in) :: beam_config
     type(interaction_t), pointer :: sf_chain_int, trace_int, src_int
     class(prc_core_t), intent(in) :: core
     type(var_list_t), intent(in), target :: process_var_list
     type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
     type(state_matrix_t), pointer :: state_matrix
     type(flavor_t), dimension(:), allocatable :: flv_int, flv_src, f_in, f_out
     integer :: n_in, n_vir, n_out, n_tot
     integer :: i, j
     term%config => config
     if (config%rearrange) then
        ! rearrangement of seed to hard kinematics not implemented yet
        ! allocate k_term distinct from k_seed as needed.
     else
        ! here, k_term trivially accesses k_seed via pointers
        call term%k_term%init_ptr (k_seed)
```

```
   end if
   allocate (term%amp (config%n_allowed))
   term%int_hard = config%int
   allocate (term%p_hard (interaction_get_n_tot (term%int_hard)))
   sf_chain_int => term%k_term%sf_chain%get_out_int_ptr ()
   n_in = interaction_get_n_in (term%int_hard)
   do j = 1, n_in
      i = term%k_term%sf_chain%get_out_i (j)
      call interaction_set_source_link (term%int_hard, j, sf_chain_int, i)
   end do
   if (config%rearrange) then
      ! rearrangement hard to effective kinematics not implemented yet
      ! should use term%config%int_eff as template
      ! allocate distinct sf_chain in term%connected as needed
   else
      ! here, int_hard and sf_chain are trivially accessed via pointers
      call term%isolated%init (term%k_term%sf_chain, term%int_hard)
   end if
   allocate (mask_in (n_in))
   mask_in = term%k_term%sf_chain%get_out_mask ()
   call term%isolated%setup_square_trace (core, mask_in, term%config%col)
   call term%connected%setup_connected_trace (term%isolated)
   associate (int_eff => term%isolated%int_eff)
     state_matrix => interaction_get_state_matrix_ptr (int_eff)
     n_tot = interaction_get_n_tot  (int_eff)
     allocate (flv_int (n_tot))
     flv_int = quantum_numbers_get_flavor &
          (state_matrix_get_quantum_numbers (state_matrix, 1))
     allocate (f_in (n_in))
     f_in = flv_int(1:n_in)
     deallocate (flv_int)
   end associate
   trace_int => evaluator_get_int_ptr (term%connected%trace)
   n_in = interaction_get_n_in (trace_int)
   n_vir = interaction_get_n_vir (trace_int)
   n_out = interaction_get_n_out (trace_int)
   allocate (f_out (n_out))
   do j = 1, n_out
      call interaction_find_source (trace_int, n_in + n_vir + j, src_int, i)
      if (associated (src_int)) then
         state_matrix => interaction_get_state_matrix_ptr (src_int)
         allocate (flv_src (interaction_get_n_tot (src_int)))
         flv_src = quantum_numbers_get_flavor &
              (state_matrix_get_quantum_numbers (state_matrix, 1))
         f_out(j) = flv_src(i)
         deallocate (flv_src)
      end if
   end do
   call term%connected%setup_subevt (term%isolated%sf_chain_eff, &
        beam_config%data%flv, f_in, f_out)
   call term%connected%setup_var_list (process_var_list, beam_config%data)
 end subroutine term_instance_init
```

For initializing the expressions, we need the local variable list and the parse

trees.

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: setup_expressions => term_instance_setup_expressions
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_setup_expressions (term, meta, config)
      class(term_instance_t), intent(inout), target :: term
      type(process_metadata_t), intent(in), target :: meta
      type(process_config_data_t), intent(in) :: config
      call term%connected%setup_expressions ( &
           config%pn_cuts, &
           config%pn_scale, &
           config%pn_fac_scale, &
           config%pn_ren_scale, &
           config%pn_weight)
    end subroutine term_instance_setup_expressions
```

Prepare the extra evaluators that we need for processing events.
    The quantum numbers mask of the incoming particle

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: setup_event_data => term_instance_setup_event_data
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_setup_event_data (term, core, model)
      class(term_instance_t), intent(inout), target :: term
      class(prc_core_t), intent(in) :: core
      type(model_t), intent(in), target :: model
      integer :: n_in
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
      n_in = interaction_get_n_in (term%int_hard)
      allocate (mask_in (n_in))
      mask_in = term%k_term%sf_chain%get_out_mask ()
      call term%isolated%setup_square_matrix (core, model, mask_in, &
           term%config%col)
      call term%isolated%setup_square_flows (core, model, mask_in)
      call term%connected%setup_connected_matrix (term%isolated)
      call term%connected%setup_connected_flows (term%isolated)
    end subroutine term_instance_setup_event_data
```

Reset the term instance: clear the parton-state expressions and deactivate.

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: reset => term_instance_reset
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_reset (term)
      class(term_instance_t), intent(inout) :: term
      call term%connected%reset_expressions ()
      term%active = .false.
    end subroutine term_instance_reset
```

Complete the kinematics computation for the effective parton states.
    We assume that the `compute_hard_kinematics` method of the process component instance has already been called, so the `int_hard` contains the correct

2061

hard kinematics. The duty of this procedure is first to compute the effective kinematics and store this in the `int_eff` effective interaction inside the `isolated` parton state. The effective kinematics may differ from the kinematics in the hard interaction. It may involve parton recombination or parton splitting. The `rearrange_partons` method is responsible for this part.

We may also call a method to compute the effective structure-function chain at this point. This is not implemented yet.

In the simple case that no rearrangement is necessary, as indicated by the `rearrange` flag, the effective interaction is a pointer to the hard interaction, and we can skip the rearrangement method. Similarly for the effective structure-function chain. (If we have an algorithm that uses rarrangement, it should evaluate `k_term` explicitly.)

The final step of kinematics setup is to transfer the effective kinematics to the evaluators and to the `subevt`.

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: compute_eff_kinematics => &
         term_instance_compute_eff_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_compute_eff_kinematics (term, component)
      class(term_instance_t), intent(inout) :: term
      type(component_instance_t), dimension(:), intent(inout) :: component
      integer :: i_component, i_term
      term%checked = .false.
      term%passed = .false.
      if (term%config%rearrange) then
         ! should evaluate k_term first if allocated separately, not impl. yet
         i_component = term%config%i_component
         i_term = term%config%i_term
         associate (core => component(i_component)%config%core)
           call core%compute_eff_kinematics &
                (i_term, term%int_hard, term%isolated%int_eff, &
                component(i_component)%tmp)
         end associate
      end if
      call term%isolated%receive_kinematics ()
      call term%connected%receive_kinematics ()
    end subroutine term_instance_compute_eff_kinematics
```

Inverse. Reconstruct the connected state from the momenta in the trace evaluator (which we assume to be set), then reconstruct the isolated state as far as possible. The second part finalizes the momentum configuration, using the incoming seed momenta

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: recover_hard_kinematics => &
         term_instance_recover_hard_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_recover_hard_kinematics (term, component)
      class(term_instance_t), intent(inout) :: term
      type(component_instance_t), dimension(:), intent(inout) :: component
      term%checked = .false.
      term%passed = .false.
```

```
        call term%connected%send_kinematics ()
        call term%isolated%send_kinematics ()
     end subroutine term_instance_recover_hard_kinematics
```

Check the term whether it passes cuts and, if successful, evaluate scales and weights. The factorization scale is also given to the term kinematics, enabling structure-function evaluation.

⟨*Processes: term instance: TBP*⟩+≡
```
     procedure :: evaluate_expressions => &
          term_instance_evaluate_expressions
```

⟨*Processes: procedures*⟩+≡
```
     subroutine term_instance_evaluate_expressions (term)
       class(term_instance_t), intent(inout) :: term
       call term%connected%evaluate_expressions (term%passed, &
            term%scale, term%fac_scale, term%ren_scale, term%weight)
       term%checked = .true.
     end subroutine term_instance_evaluate_expressions
```

Evaluate the trace: first evaluate the hard interaction, then the trace evaluator. We use the `evaluate_interaction` method of the process component which generated this term. The `subevt` and cut expressions are not yet filled.

The `component` argument is intent(inout) because the `compute_amplitude` method may modify the `tmp` workspace object.

⟨*Processes: term instance: TBP*⟩+≡
```
     procedure :: evaluate_interaction => term_instance_evaluate_interaction
```

⟨*Processes: procedures*⟩+≡
```
     subroutine term_instance_evaluate_interaction (term, component)
       class(term_instance_t), intent(inout) :: term
       type(component_instance_t), dimension(:), intent(inout) :: component
       integer :: i_component, i_term, i
       i_component = term%config%i_component
       i_term = term%config%i_term
       term%p_hard = interaction_get_momenta (term%int_hard)
       associate (core => component(i_component)%config%core)
         do i = 1, term%config%n_allowed
            term%amp(i) = core%compute_amplitude (i_term, term%p_hard, &
                 term%config%flv(i), term%config%hel(i), term%config%col(i), &
                 term%fac_scale, term%ren_scale, &
                 component(i_component)%tmp)
         end do
         call interaction_set_matrix_element (term%int_hard, term%amp)
       end associate
     end subroutine term_instance_evaluate_interaction
```

Evaluate the trace. First evaluate the structure-function chain (i.e., the density matrix of the incoming partons). Do this twice, in case the sf-chain instances within `k_term` and `isolated` differ. Next, evaluate the hard interaction, then compute the convolution with the initial state.

⟨*Processes: term instance: TBP*⟩+≡
```
     procedure :: evaluate_trace => term_instance_evaluate_trace
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_evaluate_trace (term)
      class(term_instance_t), intent(inout) :: term
      call term%k_term%evaluate_sf_chain (term%fac_scale)
      call term%isolated%evaluate_sf_chain (term%fac_scale)
      call term%isolated%evaluate_trace ()
      call term%connected%evaluate_trace ()
    end subroutine term_instance_evaluate_trace
```

Evaluate the extra data that we need for processing the object as a physical
event.

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: evaluate_event_data => term_instance_evaluate_event_data
```

⟨*Processes: procedures*⟩+≡
```
    subroutine term_instance_evaluate_event_data (term)
      class(term_instance_t), intent(inout) :: term
      call term%isolated%evaluate_event_data ()
      call term%connected%evaluate_event_data ()
    end subroutine term_instance_evaluate_event_data
```

Return data that might be useful for external processing. The factorization
scale:

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: get_fac_scale => term_instance_get_fac_scale
```

⟨*Processes: procedures*⟩+≡
```
    function term_instance_get_fac_scale (term) result (fac_scale)
      class(term_instance_t), intent(in) :: term
      real(default) :: fac_scale
      fac_scale = term%fac_scale
    end function term_instance_get_fac_scale
```

We take the strong coupling from the process core. The value is calculated
when a new event is requested, so we should call it only after the event has been
evaluated. If it is not available there (a negative number is returned), we take
the value stored in the term configuration, which should be determined by the
model. If the model does not provide a value, the result is zero.

⟨*Processes: term instance: TBP*⟩+≡
```
    procedure :: get_alpha_s => term_instance_get_alpha_s
```

⟨*Processes: procedures*⟩+≡
```
    function term_instance_get_alpha_s (term, component) result (alpha_s)
      class(term_instance_t), intent(in) :: term
      type(component_instance_t), dimension(:), intent(in) :: component
      real(default) :: alpha_s
      integer :: i_component
      i_component = term%config%i_component
      associate (core => component(i_component)%config%core)
        alpha_s = core%get_alpha_s (component(i_component)%tmp)
      end associate
      if (alpha_s < 0)  alpha_s = term%config%alpha_s
    end function term_instance_get_alpha_s
```

MC parameter set and MCI instance

For each process component that is associated with a multi-channel integration (MCI) object, the `mci_work_t` object contains the currently active parameter set. It also holds the implementation of the `mci_instance_t` that the integrator needs for doing its work.

⟨*Processes: types*⟩+≡
```
type :: mci_work_t
   type(process_mci_entry_t), pointer :: config => null ()
   real(default), dimension(:), allocatable :: x
   class(mci_instance_t), pointer :: mci => null ()
   type(process_counter_t) :: counter
 contains
 ⟨Processes: mci work: TBP⟩
end type mci_work_t
```

First write configuration data, then the current values.

⟨*Processes: mci work: TBP*⟩≡
```
procedure :: write => mci_work_write
```

⟨*Processes: procedures*⟩+≡
```
subroutine mci_work_write (mci_work, unit, testflag)
  class(mci_work_t), intent(in) :: mci_work
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u, i
  u = output_unit (unit)
  write (u, "(1x,A,I0,A)")  "Active MCI instance #", &
       mci_work%config%i_mci, " ="
  write (u, "(2x)", advance="no")
  do i = 1, mci_work%config%n_par
     write (u, "(1x,F7.5)", advance="no")  mci_work%x(i)
     if (i == mci_work%config%n_par_sf) &
          write (u, "(1x,'|')", advance="no")
  end do
  write (u, *)
  if (associated (mci_work%mci)) then
     call mci_work%mci%write (u, pacify = testflag)
     call mci_work%counter%write (u)
  end if
end subroutine mci_work_write
```

The `mci` component may require finalization.

⟨*Processes: mci work: TBP*⟩+≡
```
procedure :: final => mci_work_final
```

⟨*Processes: procedures*⟩+≡
```
subroutine mci_work_final (mci_work)
  class(mci_work_t), intent(inout) :: mci_work
  if (associated (mci_work%mci)) then
     call mci_work%mci%final ()
     deallocate (mci_work%mci)
  end if
end subroutine mci_work_final
```

Initialize with the maximum length that we will need. Contents are not initialized.

The integrator inside the `mci_entry` object is responsible for allocating and initializing its own instance, which is referred to by a pointer in the `mci_work` object.

⟨*Processes: mci work: TBP*⟩+≡
```
    procedure :: init => mci_work_init
```

⟨*Processes: procedures*⟩+≡
```
    subroutine mci_work_init (mci_work, mci_entry)
      class(mci_work_t), intent(out) :: mci_work
      type(process_mci_entry_t), intent(in), target :: mci_entry
      mci_work%config => mci_entry
      allocate (mci_work%x (mci_entry%n_par))
      if (allocated (mci_entry%mci)) then
         call mci_entry%mci%allocate_instance (mci_work%mci)
         call mci_work%mci%init (mci_entry%mci)
      end if
    end subroutine mci_work_init
```

Set parameters explicitly, either all at once, or separately for the structure-function and process parts.

⟨*Processes: mci work: TBP*⟩+≡
```
    procedure :: set => mci_work_set
    procedure :: set_x_strfun => mci_work_set_x_strfun
    procedure :: set_x_process => mci_work_set_x_process
```

⟨*Processes: procedures*⟩+≡
```
    subroutine mci_work_set (mci_work, x)
      class(mci_work_t), intent(inout) :: mci_work
      real(default), dimension(:), intent(in) :: x
      mci_work%x = x
    end subroutine mci_work_set

    subroutine mci_work_set_x_strfun (mci_work, x)
      class(mci_work_t), intent(inout) :: mci_work
      real(default), dimension(:), intent(in) :: x
      mci_work%x(1 : mci_work%config%n_par_sf) = x
    end subroutine mci_work_set_x_strfun

    subroutine mci_work_set_x_process (mci_work, x)
      class(mci_work_t), intent(inout) :: mci_work
      real(default), dimension(:), intent(in) :: x
      mci_work%x(mci_work%config%n_par_sf + 1 : mci_work%config%n_par) = x
    end subroutine mci_work_set_x_process
```

Return the array of active components, i.e., those that correspond to the currently selected MC parameter set.

⟨*Processes: mci work: TBP*⟩+≡
```
    procedure :: get_active_components => mci_work_get_active_components
```

2066

⟨*Processes: procedures*⟩+≡
```
    function mci_work_get_active_components (mci_work) result (i_component)
      class(mci_work_t), intent(in) :: mci_work
      integer, dimension(:), allocatable :: i_component
      allocate (i_component (size (mci_work%config%i_component)))
      i_component = mci_work%config%i_component
    end function mci_work_get_active_components
```

Return the active parameters as a simple array with correct length. Do this
separately for the structure-function parameters and the process parameters.
⟨*Processes: mci work: TBP*⟩+≡
```
    procedure :: get_x_strfun => mci_work_get_x_strfun
    procedure :: get_x_process => mci_work_get_x_process
```
⟨*Processes: procedures*⟩+≡
```
    function mci_work_get_x_strfun (mci_work) result (x)
      class(mci_work_t), intent(in) :: mci_work
      real(default), dimension(mci_work%config%n_par_sf) :: x
      x = mci_work%x(1 : mci_work%config%n_par_sf)
    end function mci_work_get_x_strfun

    function mci_work_get_x_process (mci_work) result (x)
      class(mci_work_t), intent(in) :: mci_work
      real(default), dimension(mci_work%config%n_par_phs) :: x
      x = mci_work%x(mci_work%config%n_par_sf + 1 : mci_work%config%n_par)
    end function mci_work_get_x_process
```

Initialize and finalize event generation for the specified MCI entry. This also
resets the counter.
⟨*Processes: mci work: TBP*⟩+≡
```
    procedure :: init_simulation => mci_work_init_simulation
    procedure :: final_simulation => mci_work_final_simulation
```
⟨*Processes: procedures*⟩+≡
```
    subroutine mci_work_init_simulation (mci_work, safety_factor)
      class(mci_work_t), intent(inout) :: mci_work
      real(default), intent(in), optional :: safety_factor
      call mci_work%mci%init_simulation (safety_factor)
      call mci_work%counter%reset ()
    end subroutine mci_work_init_simulation

    subroutine mci_work_final_simulation (mci_work)
      class(mci_work_t), intent(inout) :: mci_work
      call mci_work%mci%final_simulation ()
    end subroutine mci_work_final_simulation
```

Counter.
⟨*Processes: mci work: TBP*⟩+≡
```
    procedure :: reset_counter => mci_work_reset_counter
    procedure :: record_call => mci_work_record_call
    procedure :: get_counter => mci_work_get_counter
```

2067

⟨*Processes: procedures*⟩+≡
```
    subroutine mci_work_reset_counter (mci_work)
      class(mci_work_t), intent(inout) :: mci_work
      call mci_work%counter%reset ()
    end subroutine mci_work_reset_counter

    subroutine mci_work_record_call (mci_work, status)
      class(mci_work_t), intent(inout) :: mci_work
      integer, intent(in) :: status
      call mci_work%counter%record (status)
    end subroutine mci_work_record_call

    function mci_work_get_counter (mci_work) result (counter)
      class(mci_work_t), intent(in) :: mci_work
      type(process_counter_t) :: counter
      counter = mci_work%counter
    end function mci_work_get_counter
```

The process instance

A process instance contains all process data that depend on the sampling point and thus change often. In essence, it is an event record at the elementary (parton) level. We do not call it such, to avoid confusion with the actual event records. If decays are involved, the latter are compositions of several elementary processes (i.e., their instances).

We implement the process instance as an extension of the `mci_sampler_t` that we need for computing integrals and generate events.

The base type contains: the `integrand`, the `selected_channel`, the two-dimensional array `x` of parameters, and the one-dimensional array `f` of Jacobians. These subobjects are public and used for communicating with the multichannel integrator.

The `process` pointer accesses the process of which this record is an instance. It is required whenever the calculation needs invariant configuration data, therefore the process should stay in memory for the whole lifetime of its instances.

The `evaluation_status` code is used to check the current status. In particular, failure at various stages is recorded there.

The `count` object records process evaluations, broken down according to status.

The `sqme` value is the single real number that results from evaluating and tracing the kinematics and matrix elements. This is the number that is handed over to an integration routine.

The `weight` value is the event weight. It is defined when an event has been generated from the process instance, either weighted or unweighted. The value is the `sqme` value times Jacobian weights from the integration, or unity, respectively.

The `i_mci` index chooses a subset of components that are associated with a common parameter set and integrator, i.e., that are added coherently.

The `sf_chain` subobject is a realization of the beam and structure-function configuration in the `process` object. It is not used for calculation directly

but serves as the template for the sf-chain instances that are contained in the `component` objects.

The `component` subobjects determine the state of each component.

The `term` subobjects are workspace for evaluating kinematics, matrix elements, cuts etc.

The `mci_work` subobject contains the array of real input parameters (random numbers) that generates the kinematical point. It also contains the workspace for the MC integrators. The active entry of the `mci_work` array is selected by the `i_mci` index above.

⟨*Processes: public*⟩+≡
```
    public :: process_instance_t
```

⟨*Processes: types*⟩+≡
```
    type, extends (mci_sampler_t) :: process_instance_t
       type(process_t), pointer :: process => null ()
       integer :: evaluation_status = STAT_UNDEFINED
       real(default) :: sqme = 0
       real(default) :: weight = 0
       real(default) :: excess = 0
       integer :: i_mci = 0
       integer :: selected_channel = 0
       type(sf_chain_t) :: sf_chain
       type(component_instance_t), dimension(:), allocatable :: component
       type(term_instance_t), dimension(:), allocatable :: term
       type(mci_work_t), dimension(:), allocatable :: mci_work
     contains
     ⟨Processes: process instance: TBP⟩
    end type process_instance_t
```

The output routine contains a header with the most relevant information about the process, copied from `process_metadata_write`. We mark the active components by an asterisk.

The next section is the MC parameter input. The following sections are written only if the evaluation status is beyond setting the parameters, or if the `verbose` option is set.

⟨*Processes: process instance: TBP*⟩≡
```
    procedure :: write_header => process_instance_write_header
    procedure :: write => process_instance_write
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_write_header (object, unit, testflag)
      class(process_instance_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u, i
      u = output_unit (unit)
      call write_separator_double (u)
      if (associated (object%process)) then
        associate (meta => object%process%meta)
          select case (meta%type)
          case (PRC_UNKNOWN)
             write (u, "(1x,A)") "Process instance [undefined]"
             return
```

```
      case (PRC_DECAY)
         write (u, "(1x,A)", advance="no") "Process instance [decay]:"
      case (PRC_SCATTERING)
         write (u, "(1x,A)", advance="no") "Process instance [scattering]:"
      case default
         call msg_bug ("process_instance_write: undefined process type")
      end select
      write (u, "(1x,A,A,A)") "'", char (meta%id), "'"
      write (u, "(3x,A,A,A)") "Run ID = '", char (meta%run_id), "'"
      if (allocated (meta%component_id)) then
         write (u, "(3x,A)")  "Process components:"
         do i = 1, size (meta%component_id)
            if (object%component(i)%active) then
               write (u, "(3x,'*')", advance="no")
            else
               write (u, "(4x)", advance="no")
            end if
            write (u, "(1x,I0,9A)")  i, ": '", &
                  char (meta%component_id (i)), "':   ", &
                  char (meta%component_description (i))
         end do
      end if
   end associate
else
   write (u, "(1x,A)") "Process instance [undefined process]"
   return
end if
write (u, "(3x,A)", advance = "no")  "status = "
select case (object%evaluation_status)
case (STAT_INITIAL);          write (u, "(A)")  "initialized"
case (STAT_ACTIVATED);        write (u, "(A)")  "activated"
case (STAT_BEAM_MOMENTA);     write (u, "(A)")  "beam momenta set"
case (STAT_FAILED_KINEMATICS); write (u, "(A)")  "failed kinematics"
case (STAT_SEED_KINEMATICS);  write (u, "(A)")  "seed kinematics"
case (STAT_HARD_KINEMATICS);  write (u, "(A)")  "hard kinematics"
case (STAT_EFF_KINEMATICS);   write (u, "(A)")  "effective kinematics"
case (STAT_FAILED_CUTS);      write (u, "(A)")  "failed cuts"
case (STAT_PASSED_CUTS);      write (u, "(A)")  "passed cuts"
case (STAT_EVALUATED_TRACE);  write (u, "(A)")  "evaluated trace"
   call write_separator (u)
   write (u, "(3x,A,ES19.12)")  "sqme  = ", object%sqme
case (STAT_EVENT_COMPLETE);   write (u, "(A)")  "event complete"
   call write_separator (u)
   write (u, "(3x,A,ES19.12)")  "sqme  = ", object%sqme
   write (u, "(3x,A,ES19.12)")  "weight = ", object%weight
   if (object%excess /= 0) &
        write (u, "(3x,A,ES19.12)")  "excess = ", object%excess
case default;                 write (u, "(A)")  "undefined"
end select
if (object%i_mci /= 0) then
   call write_separator (u)
   call object%mci_work(object%i_mci)%write (u, testflag)
end if
call write_separator_double (u)
```

```
      end subroutine process_instance_write_header

   subroutine process_instance_write (object, unit, testflag)
     class(process_instance_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: testflag
     integer :: u, i
     u = output_unit (unit)
     call object%write_header (u)
     if (object%evaluation_status >= STAT_BEAM_MOMENTA) then
        call object%sf_chain%write (u)
        call write_separator_double (u)
        if (object%evaluation_status >= STAT_SEED_KINEMATICS) then
           write (u, "(1x,A)") "Active components:"
           do i = 1, size (object%component)
              if (object%component(i)%active) then
                 call write_separator (u)
                 call object%component(i)%write (u, testflag)
              end if
           end do
           if (object%evaluation_status >= STAT_HARD_KINEMATICS) then
              call write_separator_double (u)
              write (u, "(1x,A)") "Active terms:"
              if (any (object%term%active)) then
                 do i = 1, size (object%term)
                    if (object%term(i)%active) then
                       call write_separator (u)
                       call object%term(i)%write (u, &
                            show_eff_state = &
                            object%evaluation_status >= STAT_EFF_KINEMATICS, &
                            testflag = testflag)
                    end if
                 end do
              end if
           end if
           call write_separator_double (u)
        end if
     end if
   end subroutine process_instance_write
```

Finalize all subobjects that may contain allocated pointers.

⟨*Processes: process instance: TBP*⟩+≡
```
   procedure :: final => process_instance_final
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_instance_final (instance)
     class(process_instance_t), intent(inout) :: instance
     integer :: i
     if (allocated (instance%mci_work)) then
        do i = 1, size (instance%mci_work)
           call instance%mci_work(i)%final ()
        end do
     end if
     call instance%sf_chain%final ()
```

```
      if (allocated (instance%component)) then
         do i = 1, size (instance%component)
            call instance%component(i)%final ()
         end do
      end if
      if (allocated (instance%term)) then
         do i = 1, size (instance%term)
            call instance%term(i)%final ()
         end do
      end if
      instance%evaluation_status = STAT_UNDEFINED
   end subroutine process_instance_final
```

Revert the process instance to initial state. We do not deallocate anything, just
reset the state index and deactivate all components and terms.

We do not reset the choice of the MCI set `i_mci` unless this is required
explicitly.

⟨*Processes: process instance: TBP*⟩+≡
```
      procedure :: reset => process_instance_reset
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_instance_reset (instance, reset_mci)
     class(process_instance_t), intent(inout) :: instance
     logical, intent(in), optional :: reset_mci
     integer :: i
     instance%component%active = .false.
     do i = 1, size (instance%term)
        call instance%term(i)%reset ()
     end do
     instance%term%checked = .false.
     instance%term%passed = .false.
     if (present (reset_mci)) then
        if (reset_mci)  instance%i_mci = 0
     end if
     instance%selected_channel = 0
     instance%evaluation_status = STAT_INITIAL
   end subroutine process_instance_reset
```

Activate the components and terms that correspond to a currently selected MCI
parameter set.

⟨*Processes: process instance: TBP*⟩+≡
```
      procedure :: activate => process_instance_activate
```

⟨*Processes: procedures*⟩+≡
```
   subroutine process_instance_activate (instance)
     class(process_instance_t), intent(inout) :: instance
     integer :: i, j
     associate (mci_work => instance%mci_work(instance%i_mci))
       instance%component(mci_work%get_active_components ())%active &
            = .true.
       do i = 1, size (instance%component)
          associate (component => instance%component(i))
            if (component%active) then
```

```
            do j = 1, size (component%config%i_term)
               instance%term(component%config%i_term(j))%active &
                     = .true.
            end do
         end if
       end associate
     end do
   end associate
   instance%evaluation_status = STAT_ACTIVATED
 end subroutine process_instance_activate
```

Initialization connects the instance with a process. All initial information is transferred from the process object. The process object contains templates for the interaction subobjects (beam and term), but no evaluators. The initialization routine creates evaluators for the matrix element trace, other evaluators are left untouched.

The `instance` object must have the `target` attribute (also in any caller) since the initialization routine assigns various pointers to subobject of `instance`.

⟨*Processes: process instance: TBP*⟩+≡

```
   procedure :: init => process_instance_init
```

⟨*Processes: procedures*⟩+≡

```
   subroutine process_instance_init (instance, process)
     class(process_instance_t), intent(out), target :: instance
     type(process_t), intent(in), target :: process
     integer :: i, i_component
     instance%process => process
     call instance%setup_sf_chain (process%beam_config)
     allocate (instance%mci_work (process%config%n_mci))
     do i = 1, size (instance%mci_work)
        call instance%mci_work(i)%init (process%mci_entry(i))
     end do
     allocate (instance%component (process%config%n_components))
     do i_component = 1, size (instance%component)
        if (process%component(i_component)%active) then
           associate (component => instance%component(i_component))
             call component%init (process%component(i_component))
             call component%setup_kinematics &
                  (instance%sf_chain, process%beam_config)
           end associate
        end if
     end do
     allocate (instance%term (process%config%n_terms))
     do i = 1, size (instance%term)
        associate (term => instance%term(i))
          i_component = process%term(i)%i_component
          if (i_component /= 0) then
             associate (component => instance%component(i_component))
               call term%init (process%term(i), &
                    component%k_seed, &
                    process%beam_config, &
                    process%component(i_component)%core, &
                    process%meta%var_list)
```

```
               call term%setup_expressions (process%meta, process%config)
             end associate
          end if
       end associate
    end do
    instance%evaluation_status = STAT_INITIAL
  end subroutine process_instance_init
```

Subroutine of the initialization above: initialize the beam and structure-function chain template. We establish pointers to the configuration data, so `beam_config` must have a `target` attribute.

The resulting chain is not used directly for calculation. It will acquire instances which are stored in the process-component instance objects.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: setup_sf_chain => process_instance_setup_sf_chain
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_setup_sf_chain (instance, config)
      class(process_instance_t), intent(inout) :: instance
      type(process_beam_config_t), intent(in), target :: config
      integer :: n_strfun
      n_strfun = config%n_strfun
      if (n_strfun /= 0) then
         call instance%sf_chain%init (config%data, config%sf)
      else
         call instance%sf_chain%init (config%data)
      end if
      if (config%sf_trace) then
         call instance%sf_chain%setup_tracing (config%sf_trace_file)
      end if
    end subroutine process_instance_setup_sf_chain
```

This initialization routine should be called only for process instances which we intend as a source for physical events. It initializes the evaluators in the parton states of the terms. They describe the (semi-)exclusive transition matrix and the distribution of color flow for the partonic process, convoluted with the beam and structure-function chain.

If the model is not provided explicitly, we may use the model instance that belongs to the process. However, an explicit model allows us to override particle settings.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: setup_event_data => process_instance_setup_event_data
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_setup_event_data (instance, model)
      class(process_instance_t), intent(inout), target :: instance
      type(model_t), intent(in), optional, target :: model
      type(model_t), pointer :: current_model
      integer :: i, i_component
      if (present (model)) then
         current_model => model
      else
         current_model => instance%process%config%model
```

```
      end if
      do i = 1, size (instance%term)
         associate (term => instance%term(i))
           if (associated (term%config)) then
              i_component = term%config%i_component
              associate (component => instance%process%component(i_component))
                call term%setup_event_data (component%core, current_model)
              end associate
           end if
         end associate
      end do
   end subroutine process_instance_setup_event_data
```

Choose a MC parameter set and the corresponding integrator.

The choice persists beyond calls of the `reset` method above. This method is automatically called here.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: choose_mci => process_instance_choose_mci
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_choose_mci (instance, i_mci)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_mci
      instance%i_mci = i_mci
      call instance%reset ()
    end subroutine process_instance_choose_mci
```

Explicitly set a MC parameter set. Works only if we are in initial state. We assume that the length of the parameter set is correct.

After setting the parameters, activate the components and terms that correspond to the chosen MC parameter set.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: set_mcpar => process_instance_set_mcpar
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_set_mcpar (instance, x)
      class(process_instance_t), intent(inout) :: instance
      real(default), dimension(:), intent(in) :: x
      if (instance%evaluation_status == STAT_INITIAL) then
         associate (mci_work => instance%mci_work(instance%i_mci))
           call mci_work%set (x)
         end associate
         call instance%activate ()
      end if
    end subroutine process_instance_set_mcpar
```

Receive the beam momentum/momenta from a source interaction. This applies to a cascade decay process instance, where the 'beam' momentum varies event by event.

The master beam momentum array is contained in the main structure function chain subobject `sf_chain`. The sf-chain instance that reside in the components will take their beam momenta from there.

The procedure transforms the instance status into `STAT_BEAM_MOMENTA`. For process instance with fixed beam, this intermediate status is skipped.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: receive_beam_momenta => process_instance_receive_beam_momenta
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_receive_beam_momenta (instance)
  class(process_instance_t), intent(inout) :: instance
  if (instance%evaluation_status >= STAT_INITIAL) then
     call instance%sf_chain%receive_beam_momenta ()
     instance%evaluation_status = STAT_BEAM_MOMENTA
  end if
end subroutine process_instance_receive_beam_momenta
```

Set the beam momentum/momenta explicitly. Otherwise, analogous to the previous procedure.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: set_beam_momenta => process_instance_set_beam_momenta
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_set_beam_momenta (instance, p)
  class(process_instance_t), intent(inout) :: instance
  type(vector4_t), dimension(:), intent(in) :: p
  if (instance%evaluation_status >= STAT_INITIAL) then
     call instance%sf_chain%set_beam_momenta (p)
     instance%evaluation_status = STAT_BEAM_MOMENTA
  end if
end subroutine process_instance_set_beam_momenta
```

Recover the initial beam momenta (those in the `sf_chain` component), given a valid (recovered) `sf_chain_instance` in one of the active components. We need to do this only if the lab frame is not the c.m. frame, otherwise those beams would be fixed anyway.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: recover_beam_momenta => process_instance_recover_beam_momenta
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_recover_beam_momenta (instance, i_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term
  integer :: i
  if (.not. instance%process%beam_config%lab_is_cm_frame) then
     if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
        i = instance%term(i_term)%config%i_component
        call instance%component(i)%return_beam_momenta ()
     end if
  end if
end subroutine process_instance_recover_beam_momenta
```

Explicitly choose MC integration channel. We assume here that the channel count is identical for all active components.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: select_channel => process_instance_select_channel
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_select_channel (instance, channel)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: channel
      instance%selected_channel = channel
    end subroutine process_instance_select_channel
```

First step of process evaluation: set up seed kinematics. That is, for each active process component, compute a momentum array from the MC input parameters.

If `skip_term` is set, we skip the component that accesses this term. We can assume that the associated data have already been recovered, and we are just computing the rest.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: compute_seed_kinematics => &
        process_instance_compute_seed_kinematics
```
⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_compute_seed_kinematics (instance, skip_term)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in), optional :: skip_term
      integer :: channel, skip_component, i
      logical :: success
      channel = instance%selected_channel
      if (channel == 0) then
         call msg_bug ("Compute seed kinematics: undefined integration channel")
      end if
      if (present (skip_term)) then
         skip_component = instance%term(skip_term)%config%i_component
      else
         skip_component = 0
      end if
      if (instance%evaluation_status >= STAT_ACTIVATED) then
         success = .true.
         do i = 1, size (instance%component)
            if (i == skip_component)  cycle
            if (instance%component(i)%active) then
               call instance%component(i)%compute_seed_kinematics &
                    (instance%mci_work(instance%i_mci), channel, success)
               if (.not. success)  exit
            end if
         end do
         if (success) then
            instance%evaluation_status = STAT_SEED_KINEMATICS
         else
            instance%evaluation_status = STAT_FAILED_KINEMATICS
         end if
      end if
    end subroutine process_instance_compute_seed_kinematics
```

Inverse: recover missing parts of the kinematics from the momentum configuration, which we know for a single term and component. Given a channel, reconstruct the MC parameter set.

⟨*Processes: process instance: TBP*⟩+≡

```
              procedure :: recover_mcpar => process_instance_recover_mcpar
⟨Processes: procedures⟩+≡
    subroutine process_instance_recover_mcpar (instance, i_term)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_term
      integer :: channel
      integer :: i
      if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
          channel = instance%selected_channel
          if (channel == 0) then
             call msg_bug ("Recover MC parameters: undefined integration channel")
          end if
          i = instance%term(i_term)%config%i_component
          call instance%component(i)%recover_mcpar &
                     (instance%mci_work(instance%i_mci), channel)
      end if
    end subroutine process_instance_recover_mcpar
```

Second step of process evaluation: compute all momenta, for all active compo-
nents, from the seed kinematics.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: compute_hard_kinematics => &
        process_instance_compute_hard_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_compute_hard_kinematics (instance, skip_term)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in), optional :: skip_term
      integer :: i
      if (instance%evaluation_status >= STAT_SEED_KINEMATICS) then
          do i = 1, size (instance%component)
             if (instance%component(i)%active) then
                call instance%component(i)% &
                        compute_hard_kinematics (instance%term, skip_term)
             end if
          end do
          instance%evaluation_status = STAT_HARD_KINEMATICS
      end if
    end subroutine process_instance_compute_hard_kinematics
```

Inverse: recover seed kinematics. We know the beam momentum configuration
and the outgoing momenta of the effective interaction, for one specific term.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: recover_seed_kinematics => &
        process_instance_recover_seed_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_recover_seed_kinematics (instance, i_term)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_term
      if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
          associate (i_component => instance%term(i_term)%config%i_component)
            call instance%component(i_component)% &
```

```
                     recover_seed_kinematics (instance%term(i_term))
            end associate
        end if
    end subroutine process_instance_recover_seed_kinematics
```

Third step of process evaluation: compute the effective momentum configurations, for all active terms, from the hard kinematics.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: compute_eff_kinematics => &
        process_instance_compute_eff_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_compute_eff_kinematics (instance, skip_term)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in), optional :: skip_term
      integer :: i
      if (instance%evaluation_status >= STAT_HARD_KINEMATICS) then
         do i = 1, size (instance%term)
            if (present (skip_term)) then
               if (i == skip_term)  cycle
            end if
            if (instance%term(i)%active) then
               call instance%term(i)% &
                    compute_eff_kinematics (instance%component)
            end if
         end do
         instance%evaluation_status = STAT_EFF_KINEMATICS
      end if
    end subroutine process_instance_compute_eff_kinematics
```

Inverse: recover the hard kinematics from effective kinematics for one term, then compute effective kinematics for the other terms.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: recover_hard_kinematics => &
        process_instance_recover_hard_kinematics
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_recover_hard_kinematics (instance, i_term)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_term
      integer :: i
      if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
         call instance%term(i_term)%recover_hard_kinematics (instance%component)
         do i = 1, size (instance%term)
            if (i /= i_term) then
               if (instance%term(i)%active) then
                  call instance%term(i)% &
                       compute_eff_kinematics (instance%component)
               end if
            end if
         end do
         instance%evaluation_status = STAT_EFF_KINEMATICS
      end if
    end subroutine process_instance_recover_hard_kinematics
```

Fourth step of process evaluation: check cuts for all terms. Where sucessful, compute any scales and weights. Otherwise, deactive the term. If any of the terms has passed, set the state to `STAT_PASSED_CUTS`.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: evaluate_expressions => &
    process_instance_evaluate_expressions
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_evaluate_expressions (instance)
  class(process_instance_t), intent(inout) :: instance
  integer :: i
  if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
      do i = 1, size (instance%term)
          if (instance%term(i)%active) then
              call instance%term(i)%evaluate_expressions ()
          end if
      end do
      if (any (instance%term%passed)) then
          instance%evaluation_status = STAT_PASSED_CUTS
      else
          instance%evaluation_status = STAT_FAILED_CUTS
      end if
  end if
end subroutine process_instance_evaluate_expressions
```

Fifth step of process evaluation: fill the parameters for the non-selected ,channels, that have not been used for seeding. We should do this after evaluating cuts, since we may save some expensive calculations if the phase space point fails the cuts.

If `skip_term` is set, we skip the component that accesses this term. We can assume that the associated data have already been recovered, and we are just computing the rest.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: compute_other_channels => &
    process_instance_compute_other_channels
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_compute_other_channels (instance, skip_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in), optional :: skip_term
  integer :: channel, skip_component, i
  channel = instance%selected_channel
  if (channel == 0) then
      call msg_bug ("Compute other channels: undefined integration channel")
  end if
  if (present (skip_term)) then
      skip_component = instance%term(skip_term)%config%i_component
  else
      skip_component = 0
  end if
  if (instance%evaluation_status >= STAT_PASSED_CUTS) then
      do i = 1, size (instance%component)
```

2080

```
              if (i == skip_component)  cycle
              if (instance%component(i)%active) then
                 call instance%component(i)%compute_other_channels &
                       (instance%mci_work(instance%i_mci), channel)
              end if
           end do
        end if
     end subroutine process_instance_compute_other_channels
```

Sixth step of process evaluation: evaluate the matrix elements, and compute the trace (summed over quantum numbers) for all terms. Finally, sum up the terms, iterating over all active process components.

⟨*Processes: process instance: TBP*⟩+≡
```
     procedure :: evaluate_trace => process_instance_evaluate_trace
```

⟨*Processes: procedures*⟩+≡
```
     subroutine process_instance_evaluate_trace (instance)
       class(process_instance_t), intent(inout) :: instance
       integer :: i
       if (instance%evaluation_status >= STAT_PASSED_CUTS) then
          do i = 1, size (instance%term)
             associate (term => instance%term(i))
               if (term%active .and. term%passed) then
                  call term%evaluate_interaction (instance%component)
                  call term%evaluate_trace ()
               end if
             end associate
          end do
          instance%sqme = 0
          do i = 1, size (instance%component)
             associate (component => instance%component(i))
               if (component%active) then
                  call component%evaluate_sqme (instance%term)
                  instance%sqme = instance%sqme + component%sqme
               end if
             end associate
          end do
          instance%evaluation_status = STAT_EVALUATED_TRACE
       else
          ! failed kinematics, failed cuts: set sqme to zero
          instance%sqme = 0
       end if
     end subroutine process_instance_evaluate_trace
```

Final step of process evaluation: evaluate the matrix elements, and compute the trace (summed over quantum numbers) for all terms. Finally, sum up the terms, iterating over all active process components.

If `weight` is provided, we already know the kinematical event weight (the MCI weight which depends on the kinematics sampling algorithm, but not on the matrix element), so we do not need to take it from the MCI record.

⟨*Processes: process instance: TBP*⟩+≡
```
     procedure :: evaluate_event_data => process_instance_evaluate_event_data
```

2081

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_evaluate_event_data (instance, weight)
      class(process_instance_t), intent(inout) :: instance
      real(default), intent(in), optional :: weight
      integer :: i
      if (instance%evaluation_status >= STAT_EVALUATED_TRACE) then
         do i = 1, size (instance%term)
            associate (term => instance%term(i))
              if (term%active .and. term%passed) then
                 call term%evaluate_event_data ()
              end if
            end associate
         end do
         if (present (weight)) then
            instance%weight = weight
         else
            instance%weight = &
                 instance%mci_work(instance%i_mci)%mci%get_event_weight ()
            instance%excess = &
                 instance%mci_work(instance%i_mci)%mci%get_event_excess ()
         end if
         instance%evaluation_status = STAT_EVENT_COMPLETE
      else
         ! failed kinematics etc.: set weight to zero
         instance%weight = 0
      end if
    end subroutine process_instance_evaluate_event_data
```

For unweighted event generation, we should reset the reported event weight to unity (signed) or zero. The latter case is appropriate for an event which failed for whatever reason.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: normalize_weight => process_instance_normalize_weight
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_normalize_weight (instance)
      class(process_instance_t), intent(inout) :: instance
      if (instance%weight /= 0) then
         instance%weight = sign (1._default, instance%weight)
      end if
    end subroutine process_instance_normalize_weight
```

This is a convenience routine that performs the computations of the steps 1 to 5 in a single step. The arguments are the input for `set_mcpar`. After this, the evaluation status should be either `STAT_FAILED_KINEMATICS`, `STAT_FAILED_CUTS` or `STAT_EVALUATED_TRACE`.

Before calling this, we should call `choose_mci`.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: evaluate_sqme => process_instance_evaluate_sqme
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_evaluate_sqme (instance, channel, x)
      class(process_instance_t), intent(inout) :: instance
```

```
      integer, intent(in) :: channel
      real(default), dimension(:), intent(in) :: x
      call instance%reset ()
      call instance%set_mcpar (x)
      call instance%select_channel (channel)
      call instance%compute_seed_kinematics ()
      call instance%compute_hard_kinematics ()
      call instance%compute_eff_kinematics ()
      call instance%evaluate_expressions ()
      call instance%compute_other_channels ()
      call instance%evaluate_trace ()
    end subroutine process_instance_evaluate_sqme
```

This is the inverse. Assuming that the final trace evaluator contains a valid momentum configuration, recover kinematics and recalculate the matrix elements and their trace.

To be precise, we first recover kinematics for the given term and associated component, then recalculate from that all other terms and active components. The `channel` is not really required to obtain the matrix element, but it allows us to reconstruct the exact MC parameter set that corresponds to the given phase space point.

Before calling this, we should call `choose_mci`.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: recover => process_instance_recover
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_recover (instance, channel, i_term, update_sqme)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: channel
      integer, intent(in) :: i_term
      logical, intent(in) :: update_sqme
      call instance%activate ()
      instance%evaluation_status = STAT_EFF_KINEMATICS
      call instance%recover_hard_kinematics (i_term)
      call instance%recover_seed_kinematics (i_term)
      call instance%select_channel (channel)
      call instance%recover_mcpar (i_term)
      call instance%recover_beam_momenta (i_term)
      call instance%compute_seed_kinematics (i_term)
      call instance%compute_hard_kinematics (i_term)
      call instance%compute_eff_kinematics (i_term)
      call instance%compute_other_channels (i_term)
      call instance%evaluate_expressions ()
      if (update_sqme)  call instance%evaluate_trace ()
    end subroutine process_instance_recover
```

The `evaluate` method is required by the `sampler_t` base type of which the process instance is an extension.

The requirement is that after the process instance is evaluated, the integrand, the selected channel, the x array, and the f Jacobian array are exposed by the `sampler_t` object.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: evaluate => process_instance_evaluate
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_evaluate (sampler, c, x_in, val, x, f)
      class(process_instance_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(out) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      call sampler%evaluate_sqme (c, x_in)
      if (sampler%is_valid ())  call sampler%fetch (val, x, f)
      call sampler%record_call ()
    end subroutine process_instance_evaluate
```

The phase-space point is valid if the event has valid kinematics and has passed the cuts.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: is_valid => process_instance_is_valid
```

⟨*Processes: procedures*⟩+≡
```
    function process_instance_is_valid (sampler) result (valid)
      class(process_instance_t), intent(in) :: sampler
      logical :: valid
      valid = sampler%evaluation_status >= STAT_PASSED_CUTS
    end function process_instance_is_valid
```

The `rebuild` method should rebuild the kinematics section out of the `x_in` parameter set. The integrand value `val` should not be computed, but is provided as input.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: rebuild => process_instance_rebuild
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_rebuild (sampler, c, x_in, val, x, f)
      class(process_instance_t), intent(inout) :: sampler
      integer, intent(in) :: c
      real(default), dimension(:), intent(in) :: x_in
      real(default), intent(in) :: val
      real(default), dimension(:,:), intent(out) :: x
      real(default), dimension(:), intent(out) :: f
      call msg_bug ("process_instance_rebuild not implemented yet")
      x = 0
      f = 0
    end subroutine process_instance_rebuild
```

This is another method required by the `sampler_t` base type: fetch the data that are relevant for the MCI record.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: fetch => process_instance_fetch
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_fetch (sampler, val, x, f)
      class(process_instance_t), intent(in) :: sampler
      real(default), intent(out) :: val
```

```
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
        integer :: i, cc
        integer :: n_channel
        val = 0
        FIND_COMPONENT: do i = 1, size (sampler%component)
           associate (component => sampler%component(i))
             if (component%active) then
                 associate (k => component%k_seed)
                   n_channel = k%n_channel
                   do cc = 1, n_channel
                      call k%get_mcpar (cc, x(:,cc))
                   end do
                   f = k%f
                   val = sampler%sqme * k%phs_factor
                 end associate
                 exit FIND_COMPONENT
             end if
           end associate
        end do FIND_COMPONENT
      end subroutine process_instance_fetch
```

Initialize and finalize event generation for the specified MCI entry.

⟨*Processes: process instance: TBP*⟩+≡
```
      procedure :: init_simulation => process_instance_init_simulation
      procedure :: final_simulation => process_instance_final_simulation
```

⟨*Processes: procedures*⟩+≡
```
      subroutine process_instance_init_simulation (instance, i_mci, safety_factor)
        class(process_instance_t), intent(inout) :: instance
        integer, intent(in) :: i_mci
        real(default), intent(in), optional :: safety_factor
        call instance%mci_work(i_mci)%init_simulation (safety_factor)
      end subroutine process_instance_init_simulation

      subroutine process_instance_final_simulation (instance, i_mci)
        class(process_instance_t), intent(inout) :: instance
        integer, intent(in) :: i_mci
        call instance%mci_work(i_mci)%final_simulation ()
      end subroutine process_instance_final_simulation
```

15.5.12 Accessing the process instance

Once the seed kinematics is complete, we can retrieve the MC input parameters
for all channels, not just the seed channel.

Note: We choose the first active component. This makes sense only if the
seed kinematics is identical for all active components.

⟨*Processes: process instance: TBP*⟩+≡
```
      procedure :: get_mcpar => process_instance_get_mcpar
```

⟨*Processes: procedures*⟩+≡
```
      subroutine process_instance_get_mcpar (instance, channel, x)
```

```
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: channel
      real(default), dimension(:), intent(out) :: x
      integer :: i
      if (instance%evaluation_status >= STAT_SEED_KINEMATICS) then
         do i = 1, size (instance%component)
            if (instance%component(i)%active) then
               call instance%component(i)%k_seed%get_mcpar (channel, x)
               return
            end if
         end do
         call msg_bug ("Process instance: get_mcpar: no active channels")
      else
         call msg_bug ("Process instance: get_mcpar: no seed kinematics")
      end if
    end subroutine process_instance_get_mcpar
```

Return true if the sqme value is known. This also implies that the event is
kinematically valid and has passed all cuts.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: has_evaluated_trace => process_instance_has_evaluated_trace
```

⟨*Processes: procedures*⟩+≡
```
    function process_instance_has_evaluated_trace (instance) result (flag)
      class(process_instance_t), intent(in) :: instance
      logical :: flag
      flag = instance%evaluation_status >= STAT_EVALUATED_TRACE
    end function process_instance_has_evaluated_trace
```

Return true if the event is complete. In particular, the event must be kinemat-
ically valid, passed all cuts, and the event data have been computed.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: is_complete_event => process_instance_is_complete_event
```

⟨*Processes: procedures*⟩+≡
```
    function process_instance_is_complete_event (instance) result (flag)
      class(process_instance_t), intent(in) :: instance
      logical :: flag
      flag = instance%evaluation_status >= STAT_EVENT_COMPLETE
    end function process_instance_is_complete_event
```

Select a term for the process instance which is to provide the event record.

    Note: this should be done using random numbers and applying probabilities
for the various terms and components that belong to the selected MCI group.
The current implementation simply selects the first term for the first active
component.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: select_i_term => process_instance_select_i_term
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_select_i_term (instance, i_term)
      class(process_instance_t), intent(in) :: instance
      integer, intent(out) :: i_term
```

```
    integer :: i_mci, i_component
    i_mci = instance%i_mci
    i_component = instance%process%mci_entry(i_mci)%i_component(1)
    i_term = instance%process%component(i_component)%i_term(1)
  end subroutine process_instance_select_i_term
```

Return pointer to the master beam interaction.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: get_beam_int_ptr => process_instance_get_beam_int_ptr
```

⟨*Processes: procedures*⟩+≡
```
    function process_instance_get_beam_int_ptr (instance) result (ptr)
      class(process_instance_t), intent(in), target :: instance
      type(interaction_t), pointer :: ptr
      ptr => instance%sf_chain%get_beam_int_ptr ()
    end function process_instance_get_beam_int_ptr
```

Return pointers to the matrix and flows interactions, given a term index.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: get_trace_int_ptr => process_instance_get_trace_int_ptr
    procedure :: get_matrix_int_ptr => process_instance_get_matrix_int_ptr
    procedure :: get_flows_int_ptr => process_instance_get_flows_int_ptr
```

⟨*Processes: procedures*⟩+≡
```
    function process_instance_get_trace_int_ptr (instance, i_term) result (ptr)
      class(process_instance_t), intent(in), target :: instance
      integer, intent(in) :: i_term
      type(interaction_t), pointer :: ptr
      ptr => instance%term(i_term)%connected%get_trace_int_ptr ()
    end function process_instance_get_trace_int_ptr

    function process_instance_get_matrix_int_ptr (instance, i_term) result (ptr)
      class(process_instance_t), intent(in), target :: instance
      integer, intent(in) :: i_term
      type(interaction_t), pointer :: ptr
      ptr => instance%term(i_term)%connected%get_matrix_int_ptr ()
    end function process_instance_get_matrix_int_ptr

    function process_instance_get_flows_int_ptr (instance, i_term) result (ptr)
      class(process_instance_t), intent(in), target :: instance
      integer, intent(in) :: i_term
      type(interaction_t), pointer :: ptr
      ptr => instance%term(i_term)%connected%get_flows_int_ptr ()
    end function process_instance_get_flows_int_ptr
```

Return pointers to the parton states of a selected term.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: get_isolated_state_ptr => &
        process_instance_get_isolated_state_ptr
    procedure :: get_connected_state_ptr => &
        process_instance_get_connected_state_ptr
```

2087

⟨*Processes: procedures*⟩+≡
```
    function process_instance_get_isolated_state_ptr (instance, i_term) &
        result (ptr)
      class(process_instance_t), intent(in), target :: instance
      integer, intent(in) :: i_term
      type(isolated_state_t), pointer :: ptr
      ptr => instance%term(i_term)%isolated
    end function process_instance_get_isolated_state_ptr

    function process_instance_get_connected_state_ptr (instance, i_term) &
        result (ptr)
      class(process_instance_t), intent(in), target :: instance
      integer, intent(in) :: i_term
      type(connected_state_t), pointer :: ptr
      ptr => instance%term(i_term)%connected
    end function process_instance_get_connected_state_ptr
```

Return the indices of the beam particles and incoming partons within the currently active state matrix, respectively.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: get_beam_index => process_instance_get_beam_index
    procedure :: get_in_index => process_instance_get_in_index
```

⟨*Processes: procedures*⟩+≡
```
    subroutine process_instance_get_beam_index (instance, i_term, i_beam)
      class(process_instance_t), intent(in) :: instance
      integer, intent(in) :: i_term
      integer, dimension(:), intent(out) :: i_beam
      call instance%term(i_term)%connected%get_beam_index (i_beam)
    end subroutine process_instance_get_beam_index

    subroutine process_instance_get_in_index (instance, i_term, i_in)
      class(process_instance_t), intent(in) :: instance
      integer, intent(in) :: i_term
      integer, dimension(:), intent(out) :: i_in
      call instance%term(i_term)%connected%get_in_index (i_in)
    end subroutine process_instance_get_in_index
```

Return squared matrix element and event weight, and event weight excess where applicable.

⟨*Processes: process instance: TBP*⟩+≡
```
    procedure :: get_sqme => process_instance_get_sqme
    procedure :: get_weight => process_instance_get_weight
    procedure :: get_excess => process_instance_get_excess
```

⟨*Processes: procedures*⟩+≡
```
    function process_instance_get_sqme (instance) result (sqme)
      class(process_instance_t), intent(in) :: instance
      real(default) :: sqme
      if (instance%evaluation_status >= STAT_EVALUATED_TRACE) then
         sqme = instance%sqme
      else
         sqme = 0
      end if
```

```
      end function process_instance_get_sqme

      function process_instance_get_weight (instance) result (weight)
        class(process_instance_t), intent(in) :: instance
        real(default) :: weight
        if (instance%evaluation_status >= STAT_EVENT_COMPLETE) then
           weight = instance%weight
        else
           weight = 0
        end if
      end function process_instance_get_weight

      function process_instance_get_excess (instance) result (excess)
        class(process_instance_t), intent(in) :: instance
        real(default) :: excess
        if (instance%evaluation_status >= STAT_EVENT_COMPLETE) then
           excess = instance%excess
        else
           excess = 0
        end if
      end function process_instance_get_excess
```

Return the currently selected MCI channel.

⟨*Processes: process instance: TBP*⟩+≡
```
      procedure :: get_channel => process_instance_get_channel
```

⟨*Processes: procedures*⟩+≡
```
      function process_instance_get_channel (instance) result (channel)
        class(process_instance_t), intent(in) :: instance
        integer :: channel
        channel = instance%selected_channel
      end function process_instance_get_channel
```

Return factorization scale and strong coupling. We have to select a term instance.

⟨*Processes: process instance: TBP*⟩+≡
```
      procedure :: get_fac_scale => process_instance_get_fac_scale
      procedure :: get_alpha_s => process_instance_get_alpha_s
```

⟨*Processes: procedures*⟩+≡
```
      function process_instance_get_fac_scale (instance, i_term) result (fac_scale)
        class(process_instance_t), intent(in) :: instance
        integer, intent(in) :: i_term
        real(default) :: fac_scale
        fac_scale = instance%term(i_term)%get_fac_scale ()
      end function process_instance_get_fac_scale

      function process_instance_get_alpha_s (instance, i_term) result (alpha_s)
        class(process_instance_t), intent(in) :: instance
        integer, intent(in) :: i_term
        real(default) :: alpha_s
        alpha_s = instance%term(i_term)%get_alpha_s (instance%component)
      end function process_instance_get_alpha_s
```

Counter.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: reset_counter => process_instance_reset_counter
procedure :: record_call => process_instance_record_call
procedure :: get_counter => process_instance_get_counter
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_reset_counter (process_instance)
  class(process_instance_t), intent(inout) :: process_instance
  call process_instance%mci_work(process_instance%i_mci)%reset_counter ()
end subroutine process_instance_reset_counter

subroutine process_instance_record_call (process_instance)
  class(process_instance_t), intent(inout) :: process_instance
  call process_instance%mci_work(process_instance%i_mci)%record_call &
       (process_instance%evaluation_status)
end subroutine process_instance_record_call

function process_instance_get_counter (process_instance) result (counter)
  class(process_instance_t), intent(in) :: process_instance
  type(process_counter_t) :: counter
  counter = process_instance%mci_work(process_instance%i_mci)%get_counter ()
end function process_instance_get_counter
```

15.5.13   Particle sets

Here we provide two procedures that convert the process instance from/to a particle set. The conversion applies to the trace evaluator which has no quantum-number information, thus it involves only the momenta and the parent-child relations. We keep virtual particles.

Nevertheless, it is possible to reconstruct the complete structure from a particle set. The reconstruction implies a re-evaluation of the structure function and matrix-element codes.

The `i_term` index is needed for both input and output, to select among different active trace evaluators.

In both cases, the `instance` object must be properly initialized.

⟨*Processes: process instance: TBP*⟩+≡
```
procedure :: get_trace => process_instance_get_trace
procedure :: set_trace => process_instance_set_trace
```

⟨*Processes: procedures*⟩+≡
```
subroutine process_instance_get_trace (instance, pset, i_term)
  class(process_instance_t), intent(in), target :: instance
  type(particle_set_t), intent(out) :: pset
  integer, intent(in) :: i_term
  type(interaction_t), pointer :: int
  logical :: ok
  int => instance%get_trace_int_ptr (i_term)
  call particle_set_init (pset, ok, int, int, FM_IGNORE_HELICITY, &
       [0._default, 0._default], .false., .true.)
end subroutine process_instance_get_trace
```

2090

```
subroutine process_instance_set_trace (instance, pset, i_term, recover_beams)
  class(process_instance_t), intent(inout), target :: instance
  type(particle_set_t), intent(in) :: pset
  integer, intent(in) :: i_term
  logical, intent(in), optional :: recover_beams
  type(interaction_t), pointer :: int
  integer :: n_in
  int => instance%get_trace_int_ptr (i_term)
  n_in = instance%process%get_n_in ()
  call particle_set_fill_interaction (pset, int, n_in, recover_beams)
end subroutine process_instance_set_trace
```

The `pacify` subroutine has the purpose of setting numbers to zero which are (by comparing with a `tolerance` parameter) considered equivalent with zero. We do this in some unit tests. Here, we a apply this to the phase space subobject of the process instance.

⟨*Processes: public*⟩+≡
```
public :: pacify
```

⟨*Processes: interfaces*⟩≡
```
interface pacify
   module procedure pacify_process_instance
end interface pacify
```

⟨*Processes: procedures*⟩+≡
```
subroutine pacify_process_instance (instance)
  type(process_instance_t), intent(inout) :: instance
  integer :: i
  do i = 1, size (instance%component)
     call pacify (instance%component(i)%k_seed%phs)
  end do
end subroutine pacify_process_instance
```

15.5.14   Unit tests

⟨*Processes: public*⟩+≡
```
public :: processes_test
```

⟨*Processes: tests*⟩≡
```
subroutine processes_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
  ⟨Processes: execute tests⟩
end subroutine processes_test
```

Test process type

For the following tests, we define a simple implementation of the abstract `prc_core_t`, designed such as to complement the `prc_test_t` process definition type.

Note that it is not given that the actual process is defined as `prc_test_t` type. We enforce this by calling `prc_test_create_library`. The driver component in the process core will then become of type `prc_test_t`.

⟨*Processes: public*⟩+≡
```
    public :: test_t
```

⟨*Processes: test types*⟩≡
```
    type, extends (prc_core_t) :: test_t
     contains
       procedure :: write => test_write
       procedure :: needs_mcset => test_needs_mcset
       procedure :: get_n_terms => test_get_n_terms
       procedure :: is_allowed => test_is_allowed
       procedure :: compute_hard_kinematics => test_compute_hard_kinematics
       procedure :: compute_eff_kinematics => test_compute_eff_kinematics
       procedure :: recover_kinematics => test_recover_kinematics
       procedure :: compute_amplitude => test_compute_amplitude
    end type test_t
```

⟨*Processes: tests*⟩+≡
```
    subroutine test_write (object, unit)
      class(test_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A)")  "test type implementing prc_test"
    end subroutine test_write
```

This process type always needs a MC parameter set and a single term. This only state is always allowed.

⟨*Processes: tests*⟩+≡
```
    function test_needs_mcset (object) result (flag)
      class(test_t), intent(in) :: object
      logical :: flag
      flag = .true.
    end function test_needs_mcset

    function test_get_n_terms (object) result (n)
      class(test_t), intent(in) :: object
      integer :: n
      n = 1
    end function test_get_n_terms

    function test_is_allowed (object, i_term, f, h, c) result (flag)
      class(test_t), intent(in) :: object
      integer, intent(in) :: i_term, f, h, c
      logical :: flag
      flag = .true.
    end function test_is_allowed
```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

⟨*Processes: tests*⟩+≡

```
subroutine test_compute_hard_kinematics &
    (object, p_seed, i_term, int_hard, tmp)
  class(test_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(in) :: p_seed
  integer, intent(in) :: i_term
  type(interaction_t), intent(inout) :: int_hard
  class(workspace_t), intent(inout), allocatable :: tmp
  call interaction_set_momenta (int_hard, p_seed)
end subroutine test_compute_hard_kinematics
```

This procedure is not called for `test_t`, just a placeholder.

⟨*Processes: tests*⟩+≡
```
subroutine test_compute_eff_kinematics &
    (object, i_term, int_hard, int_eff, tmp)
  class(test_t), intent(in) :: object
  integer, intent(in) :: i_term
  type(interaction_t), intent(in) :: int_hard
  type(interaction_t), intent(inout) :: int_eff
  class(workspace_t), intent(inout), allocatable :: tmp
end subroutine test_compute_eff_kinematics
```

Transfer the incoming momenta of `p_seed` directly to the effective interaction, and vice versa for the outgoing momenta.

`int_hard` is left untouched since `int_eff` is an alias (via pointer) to it.

⟨*Processes: tests*⟩+≡
```
subroutine test_recover_kinematics &
    (object, p_seed, int_hard, int_eff, tmp)
  class(test_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(inout) :: p_seed
  type(interaction_t), intent(inout) :: int_hard
  type(interaction_t), intent(inout) :: int_eff
  class(workspace_t), intent(inout), allocatable :: tmp
  integer :: n_in
  n_in = interaction_get_n_in (int_eff)
  call interaction_set_momenta (int_eff, p_seed(1:n_in), outgoing = .false.)
  p_seed(n_in+1:) = interaction_get_momenta (int_eff, outgoing = .true.)
end subroutine test_recover_kinematics
```

Compute the amplitude. The driver ignores all quantum numbers and, in fact, returns a constant. Nevertheless, we properly transfer the momentum vectors.

⟨*Processes: tests*⟩+≡
```
function test_compute_amplitude &
    (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
  class(test_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  class(workspace_t), intent(inout), allocatable, optional :: tmp
  complex(default) :: amp
  real(default), dimension(:,:), allocatable :: parray
  integer :: i, n_tot
```

```
      select type (driver => object%driver)
      type is (prc_test_t)
         if (driver%scattering) then
            n_tot = 4
         else
            n_tot = 3
         end if
         allocate (parray (0:3,n_tot))
         forall (i = 1:n_tot)  parray(:,i) = vector4_get_components (p(i))
         amp = driver%get_amplitude (parray)
      end select
    end function test_compute_amplitude
```

Write an empty process object

The most trivial test is to write an uninitialized process object.

⟨*Processes: execute tests*⟩≡
```
    call test (processes_1, "processes_1", &
         "write an empty process object", &
         u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_1 (u)
      integer, intent(in) :: u
      type(process_t) :: process

      write (u, "(A)")  "* Test output: processes_1"
      write (u, "(A)")  "*   Purpose: display an empty process object"
      write (u, "(A)")

      call process%write (.false., u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_1"

    end subroutine processes_1
```

Initialize a process object

Initialize a process and display it.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_2, "processes_2", &
         "initialize a simple process object", &
         u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_2 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
```

```
   type(os_data_t) :: os_data
   type(qcd_t) :: qcd
   class(rng_factory_t), allocatable :: rng_factory
   type(model_list_t) :: model_list
   type(process_t), allocatable :: process
   class(prc_core_t), allocatable :: core_template
   class(mci_t), allocatable :: mci_template
   class(phs_config_t), allocatable :: phs_config_template

   write (u, "(A)")  "* Test output: processes_2"
   write (u, "(A)")  "*   Purpose: initialize a simple process object"
   write (u, "(A)")

   write (u, "(A)")  "* Build and load a test library with one process"
   write (u, "(A)")

   libname = "processes2"
   procname = libname
   run_id = "run2"
   call os_data_init (os_data)
   allocate (rng_test_factory_t :: rng_factory)
   call prc_test_create_library (libname, lib)
   call syntax_model_file_init ()

   write (u, "(A)")  "* Initialize a process object"
   write (u, "(A)")

   allocate (process)
   call process%init &
        (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

   allocate (test_t :: core_template)
   allocate (phs_test_config_t :: phs_config_template)
   call process%init_component &
        (1, core_template, mci_template, phs_config_template)

   call process%setup_mci ()

   call process%write (.false., u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call process%final ()
   deallocate (process)

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: processes_2"

end subroutine processes_2
```

Compute a trivial matrix element

Initialize a process, retrieve some information and compute a matrix element.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_3, "processes_3", &
        "retrieve a trivial matrix element", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_3 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      type(process_constants_t) :: data
      type(vector4_t), dimension(:), allocatable :: p

      write (u, "(A)")  "* Test output: processes_3"
      write (u, "(A)")  "*   Purpose: create a process &
          &and compute a matrix element"
      write (u, "(A)")

      write (u, "(A)")  "* Build and load a test library with one process"
      write (u, "(A)")

      libname = "processes3"
      procname = libname
      run_id = "run3"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call syntax_model_file_init ()

      allocate (process)
      call process%init &
          (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

      allocate (test_t :: core_template)
      allocate (mci_test_t :: mci_template)
      select type (mci_template)
      type is (mci_test_t)
         call mci_template%set_dimensions (2, 2)
         call mci_template%set_divisions (100)
      end select
```

2096

```
    allocate (phs_test_config_t :: phs_config_template)
    call process%init_component &
         (1, core_template, mci_template, phs_config_template)

    write (u, "(A)")  "* Return the number of process components"
    write (u, "(A)")

    write (u, "(A,I0)")  "n_components = ", process%get_n_components ()

    write (u, "(A)")
    write (u, "(A)")  "* Return the number of flavor states"
    write (u, "(A)")

    data = process%get_constants (1)

    write (u, "(A,I0)")  "n_flv(1) = ", data%n_flv

    write (u, "(A)")
    write (u, "(A)")  "* Return the first flavor state"
    write (u, "(A)")

    write (u, "(A,4(1x,I0))")  "flv_state(1) =", data%flv_state (:,1)

    write (u, "(A)")
    write (u, "(A)")  "* Set up kinematics &
         &[arbitrary, the matrix element is constant]"

    allocate (p (4))

    write (u, "(A)")
    write (u, "(A)")  "* Retrieve the matrix element"
    write (u, "(A)")

    write (u, "(A,F5.3,' + ',F5.3,' I')")  "me (1, p, 1, 1, 1) = ", &
         process%compute_amplitude (1, 1, p, 1, 1, 1)


    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call process%final ()
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_3"

  end subroutine processes_3
```

Generate a process instance

Initialize a process and process instance, choose a sampling point and fill the process instance.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_4, "processes_4", &
         "create and fill a process instance (partonic event)", &
         u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_4 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance
      type(particle_set_t) :: pset

      write (u, "(A)")  "* Test output: processes_4"
      write (u, "(A)")  "*   Purpose: create a process &
           &and fill a process instance"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a test process"
      write (u, "(A)")

      libname = "processes4"
      procname = libname
      run_id = "run4"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call syntax_model_file_init ()

      call reset_interaction_counter ()

      allocate (process)
      call process%init &
           (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

      allocate (test_t :: core_template)
      allocate (phs_test_config_t :: phs_config_template)
      call process%init_component &
```

```
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()

write (u, "(A)")  "* Evaluate matrix element and square"
write (u, "(A)")

call process_instance%evaluate_trace ()
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Particle content:"
    write (u, "(A)")

    call write_separator (u)
    call particle_set_write (pset, u)
    call write_separator (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recover process instance"
    write (u, "(A)")

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%choose_mci (1)
    call process_instance%set_trace (pset, 1)

    call process_instance%activate ()
    process_instance%evaluation_status = STAT_EFF_KINEMATICS
    call process_instance%recover_hard_kinematics (i_term = 1)
    call process_instance%recover_seed_kinematics (i_term = 1)
    call process_instance%select_channel (1)
    call process_instance%recover_mcpar (i_term = 1)

    call process_instance%compute_seed_kinematics (skip_term = 1)
    call process_instance%compute_hard_kinematics (skip_term = 1)
    call process_instance%compute_eff_kinematics (skip_term = 1)

    call process_instance%evaluate_expressions ()
    call process_instance%compute_other_channels (skip_term = 1)
    call process_instance%evaluate_trace ()
    call process_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call particle_set_final (pset)
    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_4"

  end subroutine processes_4
```

Handle partonic cuts

Initialize a process and process instance, choose a sampling point and fill the process instance, evaluating a given cut configuration.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_5, "processes_5", &
        "handle cuts (partonic event)", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_5 (u)
      integer, intent(in) :: u
      type(string_t) :: cut_expr_text
      type(ifile_t) :: ifile
      type(stream_t) :: stream
      type(parse_tree_t) :: parse_tree
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance

      write (u, "(A)")  "* Test output: processes_5"
      write (u, "(A)")  "*   Purpose: create a process &
          &and fill a process instance"
      write (u, "(A)")

      write (u, "(A)")  "* Prepare a cut expression"
      write (u, "(A)")

      call syntax_pexpr_init ()
      cut_expr_text = "all Pt > 100 [s]"
      call ifile_append (ifile, cut_expr_text)
      call stream_init (stream, ifile)
      call parse_tree_init_lexpr (parse_tree, stream, .true.)

      write (u, "(A)")  "* Build and initialize a test process"
      write (u, "(A)")

      libname = "processes5"
      procname = libname
      run_id = "run5"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
```

2101

```
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)
call process%set_var_list (model_get_var_list_ptr (process%config%model))
call var_list_append_real &
     (process%meta%var_list, var_str ("tolerance"), 0._default)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization and set cuts"
write (u, "(A)")

call process%setup_terms ()
call process%set_cuts (parse_tree_get_root_ptr (parse_tree))
call process%write (.false., u, show_var_list=.true., show_expressions=.true.)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics and subevt, check cuts (should fail)"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
```

2102

```
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set (should succeed)"
write (u, "(A)")

call process_instance%reset ()
call process_instance%set_mcpar ([0.5_default, 0.125_default])
call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()
call process_instance%evaluate_trace ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set using convenience procedure &
     &(failure)"
write (u, "(A)")

call process_instance%evaluate_sqme (1, [0.0_default, 0.2_default])

call process_instance%write_header (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set using convenience procedure &
     &(success)"
write (u, "(A)")

call process_instance%evaluate_sqme (1, [0.1_default, 0.2_default])

call process_instance%write_header (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)

call parse_tree_final (parse_tree)
call stream_final (stream)
call ifile_final (ifile)
call syntax_pexpr_final ()

call model_list%final ()
```

```
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_5"

    end subroutine processes_5
```

Scales and such

Initialize a process and process instance, choose a sampling point and fill the process instance, evaluating a given cut configuration.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Processes: execute tests*⟩+≡
```
      call test (processes_6, "processes_6", &
           "handle scales and weight (partonic event)", &
           u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_6 (u)
      integer, intent(in) :: u
      type(string_t) :: expr_text
      type(ifile_t) :: ifile
      type(stream_t) :: stream
      type(parse_tree_t) :: pt_scale, pt_fac_scale, pt_ren_scale, pt_weight
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance

      write (u, "(A)")  "* Test output: processes_6"
      write (u, "(A)")  "*   Purpose: create a process &
           &and fill a process instance"
      write (u, "(A)")

      write (u, "(A)")  "* Prepare expressions"
      write (u, "(A)")

      call syntax_pexpr_init ()

      expr_text = "sqrts - 100 GeV"
      write (u, "(A,A)")  "scale = ", char (expr_text)
      call ifile_clear (ifile)
```

2104

```
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_scale, stream, .true.)
call stream_final (stream)

expr_text = "sqrts_hat"
write (u, "(A,A)")  "fac_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_fac_scale, stream, .true.)
call stream_final (stream)

expr_text = "eval sqrt (M2) [collect [s]]"
write (u, "(A,A)")  "ren_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_ren_scale, stream, .true.)
call stream_final (stream)

expr_text = "n_tot * n_in * n_out * (eval Phi / pi [s])"
write (u, "(A,A)")  "weight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Build and initialize a test process"
write (u, "(A)")

libname = "processes4"
procname = libname
run_id = "run4"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)
call process%set_var_list (model_get_var_list_ptr (process%config%model))

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)
```

```
      write (u, "(A)")   "* Prepare a trivial beam setup"
      write (u, "(A)")

      sqrts = 1000
      call process%setup_beams_sqrts (sqrts)
      call process%configure_phs ()
      call process%setup_mci ()

      write (u, "(A)")   "* Complete process initialization and set cuts"
      write (u, "(A)")

      call process%setup_terms ()
      call process%set_scale (parse_tree_get_root_ptr (pt_scale))
      call process%set_fac_scale (parse_tree_get_root_ptr (pt_fac_scale))
      call process%set_ren_scale (parse_tree_get_root_ptr (pt_ren_scale))
      call process%set_weight (parse_tree_get_root_ptr (pt_weight))
      call process%write (.false., u, show_expressions=.true.)

      write (u, "(A)")
      write (u, "(A)")   "* Create a process instance and evaluate"
      write (u, "(A)")

      allocate (process_instance)
      call process_instance%init (process)
      call process_instance%choose_mci (1)
      call process_instance%evaluate_sqme (1, [0.5_default, 0.125_default])

      call process_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call process_instance%final ()
      deallocate (process_instance)

      call process%final ()
      deallocate (process)

      call parse_tree_final (pt_scale)
      call parse_tree_final (pt_fac_scale)
      call parse_tree_final (pt_ren_scale)
      call parse_tree_final (pt_weight)
      call syntax_pexpr_final ()

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: processes_6"

    end subroutine processes_6
```

Structure function configuration

Configure structure functions (multi-channel) in a process object.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_7, "processes_7", &
         "process configuration with structure functions", &
         u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_7 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_channel_t), dimension(2) :: sf_channel

      write (u, "(A)")  "* Test output: processes_7"
      write (u, "(A)")  "*   Purpose: initialize a process with &
          &structure functions"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a process object"
      write (u, "(A)")

      libname = "processes7"
      procname = libname
      run_id = "run7"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call syntax_model_file_init ()

      allocate (process)
      call process%init &
          (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

      allocate (test_t :: core_template)
      allocate (phs_test_config_t :: phs_config_template)
      call process%init_component &
          (1, core_template, mci_template, phs_config_template)
```

```
      write (u, "(A)")  "* Set beam, structure functions, and mappings"
      write (u, "(A)")

      sqrts = 1000
      call process%setup_beams_sqrts (sqrts)
      call process%configure_phs ()

      model => model_list%get_model_ptr (var_str ("Test"))
      pdg_in = 25
      allocate (sf_test_data_t :: data)
      select type (data)
      type is (sf_test_data_t)
         call data%init (model, pdg_in)
      end select

      allocate (sf_config (2))
      call sf_config(1)%init ([1], data)
      call sf_config(2)%init ([2], data)
      call process%init_sf_chain (sf_config)
      deallocate (sf_config)

      call process%beam_config%allocate_sf_channels (3)

      call sf_channel(1)%init (2)
      call sf_channel(1)%activate_mapping ([1,2])
      call process%set_sf_channel (2, sf_channel(1))

      call sf_channel(2)%init (2)
      call sf_channel(2)%set_s_mapping ([1,2])
      call process%set_sf_channel (3, sf_channel(2))

      call process%setup_mci ()

      call process%write (.false., u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call process%final ()
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_7"

    end subroutine processes_7
```

Evaluating a process with structure function

Configure structure functions (single-channel) in a process object, create an instance, compute kinematics and evaluate.

Note the order of operations when setting up structure functions and phase space. The beams are first, they determine the `sqrts` value. We can also set up the chain of structure functions. We then configure the phase space. From this, we can obtain information about special configurations (resonances, etc.), which we need for allocating the possible structure-function channels (parameterizations and mappings). Finally, we match phase-space channels onto structure-function channels.

In the current example, this matching is trivial; we only have one structure-function channel.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_8, "processes_8", &
        "process evaluation with structure functions", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_8 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_channel_t) :: sf_channel
      type(particle_set_t) :: pset

      write (u, "(A)")  "* Test output: processes_8"
      write (u, "(A)")  "*   Purpose: evaluate a process with &
          &structure functions"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a process object"
      write (u, "(A)")

      libname = "processes8"
      procname = libname
      run_id = "run8"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call syntax_model_file_init ()
```

```
call reset_interaction_counter ()

allocate (process)
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts)

model => model_list%get_model_ptr (var_str ("Test"))
pdg_in = 25
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in)
end select

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call process%configure_phs ()

call process%beam_config%allocate_sf_channels (1)

call sf_channel%init (2)
call sf_channel%activate_mapping ([1,2])
call process%set_sf_channel (1, sf_channel)

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_mci ()
call process%setup_terms ()

call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
```

```
   write (u, "(A)")  "* Set up kinematics and evaluate"
   write (u, "(A)")

   call process_instance%choose_mci (1)
   call process_instance%evaluate_sqme (1, &
        [0.8_default, 0.8_default, 0.1_default, 0.2_default])
   call process_instance%write (u)

   call process_instance%get_trace (pset, 1)
   call process_instance%final ()
   deallocate (process_instance)

   write (u, "(A)")
   write (u, "(A)")  "* Particle content:"
   write (u, "(A)")

   call write_separator (u)
   call particle_set_write (pset, u)
   call write_separator (u)

   write (u, "(A)")
   write (u, "(A)")  "* Recover process instance"
   write (u, "(A)")

   call reset_interaction_counter (2)

   allocate (process_instance)
   call process_instance%init (process)

   call process_instance%choose_mci (1)
   call process_instance%set_trace (pset, 1)
   call process_instance%recover &
        (channel = 1, i_term = 1, update_sqme = .true.)
   call process_instance%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call particle_set_final (pset)

   call process_instance%final ()
   deallocate (process_instance)

   call process%final ()
   deallocate (process)

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: processes_8"

end subroutine processes_8
```

Multi-channel phase space and structure function

This is an extension of the previous example. This time, we have two distinct structure-function channels which are matched to the two distinct phase-space channels.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_9, "processes_9", &
        "multichannel kinematics and structure functions", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_9 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_channel_t) :: sf_channel
      real(default), dimension(4) :: x_saved
      type(particle_set_t) :: pset

      write (u, "(A)")  "* Test output: processes_9"
      write (u, "(A)")  "*   Purpose: evaluate a process with &
          &structure functions"
      write (u, "(A)")  "*              in a multi-channel configuration"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a process object"
      write (u, "(A)")

      libname = "processes9"
      procname = libname
      run_id = "run9"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call syntax_model_file_init ()

      call reset_interaction_counter ()

      allocate (process)
```

2112

```
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts)

model => model_list%get_model_ptr (var_str ("Test"))
pdg_in = 25
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in)
end select

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call process%configure_phs ()

call process%beam_config%allocate_sf_channels (2)

call sf_channel%init (2)
call process%set_sf_channel (1, sf_channel)

call sf_channel%init (2)
call sf_channel%activate_mapping ([1,2])
call process%set_sf_channel (2, sf_channel)

call process%component(1)%phs_config%set_sf_channel ([1, 2])

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_mci ()
call process%setup_terms ()

call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
```

```
call process_instance%init (process)

write (u, "(A)")  "* Set up kinematics in channel 1 and evaluate"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, &
     [0.8_default, 0.8_default, 0.1_default, 0.2_default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract MC input parameters"
write (u, "(A)")

write (u, "(A)")  "Channel 1:"
call process_instance%get_mcpar (1, x_saved)
write (u, "(2x,9(1x,F7.5))")  x_saved

write (u, "(A)")  "Channel 2:"
call process_instance%get_mcpar (2, x_saved)
write (u, "(2x,9(1x,F7.5))")  x_saved

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics in channel 2 and evaluate"
write (u, "(A)")

call process_instance%evaluate_sqme (2, x_saved)
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance for channel 2"
write (u, "(A)")

call reset_interaction_counter (2)

allocate (process_instance)
call process_instance%init (process)

call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1)
call process_instance%recover &
     (channel = 2, i_term = 1, update_sqme = .true.)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (pset)

call process_instance%final ()
```

2114

```
      deallocate (process_instance)

      call process%final ()
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_9"

    end subroutine processes_9
```

Event generation

Activate the MC integrator for the process object and use it to generate a single
event. Note that the test integrator does not require integration in preparation
for generating events.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_10, "processes_10", &
         "event generation", &
         u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_10 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance

      write (u, "(A)")  "* Test output: processes_10"
      write (u, "(A)")  "*   Purpose: generate events for a process without &
           &structure functions"
      write (u, "(A)")  "*            in a multi-channel configuration"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a process object"
      write (u, "(A)")

      libname = "processes10"
      procname = libname
      run_id = "run10"
```

```
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t);  call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts)
call process%configure_phs ()

call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Generate weighted event"
write (u, "(A)")

select type (mci => process%mci_entry(1)%mci)
type is (mci_test_t)
   ! This ensures that the next 'random' numbers are 0.3, 0.5, 0.7
   call mci%rng%init (3)
   ! Include the constant PHS factor in the stored maximum of the integrand
   call mci%set_max_factor (conv * twopi4 &
        / (2 * sqrt (lambda (sqrts **2, 125._default**2, 125._default**2))))
end select
```

```
      call process%generate_weighted_event (process_instance, 1)
      call process_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Generate unweighted event"
      write (u, "(A)")

      call process%generate_unweighted_event (process_instance, 1)
      select type (mci => process%mci_entry(1)%mci)
      type is (mci_test_t)
         write (u, "(A,I0)")  " Success in try ", mci%tries
         write (u, "(A)")
      end select

      call process_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call process_instance%final ()
      deallocate (process_instance)

      call process%final ()
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_10"

    end subroutine processes_10
```

Integration

Activate the MC integrator for the process object and use it to integrate over phase space.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_11, "processes_11", &
        "integration", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_11 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
```

```
type(process_t), allocatable, target :: process
class(prc_core_t), allocatable :: core_template
class(mci_t), allocatable :: mci_template
class(phs_config_t), allocatable :: phs_config_template
real(default) :: sqrts
type(process_instance_t), allocatable, target :: process_instance

write (u, "(A)")  "* Test output: processes_11"
write (u, "(A)")  "*   Purpose: integrate a process without &
     &structure functions"
write (u, "(A)")  "*            in a multi-channel configuration"
write (u, "(A)")

write (u, "(A)")  "* Build and initialize a process object"
write (u, "(A)")

libname = "processes11"
procname = libname
run_id = "run11"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)
call syntax_model_file_init ()

call reset_interaction_counter ()

allocate (process)
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t)
   call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts)
call process%configure_phs ()

call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)
```

2118

```
      write (u, "(A)")
      write (u, "(A)")  "* Create a process instance"
      write (u, "(A)")

      allocate (process_instance)
      call process_instance%init (process)

      write (u, "(A)")  "* Integrate with default test parameters"
      write (u, "(A)")

      call process%integrate (process_instance, 1, n_it=1, n_calls=10000)
      call process%final_integration (1)

      call process%write (.false., u)

      write (u, "(A)")
      write (u, "(A,ES13.7)")  " Integral divided by phs factor = ", &
           process%get_integral (1) &
           / process_instance%component(1)%k_seed%phs_factor

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call process_instance%final ()
      deallocate (process_instance)

      call process%final ()
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_11"

    end subroutine processes_11
```

Complete events

For the purpose of simplifying further tests, we implement a convenience routine that initializes a process and prepares a single event. This is a wrapup of the test `processes_10`.

⟨*Processes: public*⟩+≡
```
    public :: prepare_test_process
```

⟨*Processes: procedures*⟩+≡
```
    subroutine prepare_test_process (process, process_instance, model_list)
      type(process_t), intent(out), target :: process
      type(process_instance_t), intent(out), target :: process_instance
      type(model_list_t), intent(inout) :: model_list
      type(process_library_t), target :: lib
      type(string_t) :: libname
```

```
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      libname = "processes_test"
      procname = libname
      run_id = "run_test"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call reset_interaction_counter ()
      call process%init &
           (procname, run_id, lib, os_data, qcd, rng_factory, model_list)
      allocate (test_t :: core_template)
      allocate (mci_test_t :: mci_template)
      select type (mci_template)
      type is (mci_test_t);  call mci_template%set_divisions (100)
      end select
      allocate (phs_test_config_t :: phs_config_template)
      call process%init_component &
           (1, core_template, mci_template, phs_config_template)
      sqrts = 1000
      call process%setup_beams_sqrts (sqrts)
      call process%configure_phs ()
      call process%setup_mci ()
      call process%setup_terms ()
      call process_instance%init (process)
      select type (mci => process%mci_entry(1)%mci)
      type is (mci_test_t)
         ! This ensures that the next 'random' numbers are 0.3, 0.5, 0.7
         call mci%rng%init (3)
         ! Include the constant PHS factor in the stored maximum of the integrand
         call mci%set_max_factor (conv * twopi4 &
              / (2 * sqrt (lambda (sqrts **2, 125._default**2, 125._default**2))))
      end select
    end subroutine prepare_test_process
```

Here we do the cleanup of the process and process instance emitted by the previous routine.

⟨*Processes: public*⟩+≡
```
    public :: cleanup_test_process
```

⟨*Processes: procedures*⟩+≡
```
    subroutine cleanup_test_process (process, process_instance)
      type(process_t), intent(inout) :: process
      type(process_instance_t), intent(inout) :: process_instance
      call process_instance%final ()
      call process%final ()
    end subroutine cleanup_test_process
```

This is the actual test. Prepare the test process and event, fill all evaluators, and display the results. Use a particle set as temporary storage, read kinematics and recalculate the event.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_12, "processes_12", &
        "event post-processing", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_12 (u)
      integer, intent(in) :: u
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(particle_set_t) :: pset
      type(model_list_t) :: model_list

      write (u, "(A)")  "* Test output: processes_12"
      write (u, "(A)")  "*   Purpose: generate a complete partonic event"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Build and initialize process and process instance &
          &and generate event"
      write (u, "(A)")

      allocate (process)
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      call process%prepare_simulation (1)
      call process_instance%init_simulation (1)
      call process%generate_weighted_event (process_instance, 1)
      call process_instance%evaluate_event_data ()

      call process_instance%write (u)

      call process_instance%get_trace (pset, 1)

      call process_instance%final_simulation (1)
      call process_instance%final ()
      deallocate (process_instance)

      write (u, "(A)")
      write (u, "(A)")  "* Recover kinematics and recalculate"
      write (u, "(A)")

      call reset_interaction_counter (2)

      allocate (process_instance)
      call process_instance%init (process)
      call process_instance%setup_event_data ()
```

```
      call process_instance%choose_mci (1)
      call process_instance%set_trace (pset, 1)
      call process_instance%recover &
           (channel = 1, i_term = 1, update_sqme = .true.)

      call process%recover_event (process_instance, 1)
      call process_instance%evaluate_event_data ()

      call process_instance%write (u)


      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_12"

    end subroutine processes_12
```

Colored interaction

This test specifically checks the transformation of process data (flavor, helicity, and color) into an interaction in a process term.

We use the `test_t` process core (which has no nontrivial particles), but call only the `is_allowed` method, which always returns true.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_13, "processes_13", &
         "colored interaction", &
         u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_13 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_term_t) :: term
      class(prc_core_t), allocatable :: core

      write (u, "(A)")  "* Test output: processes_12"
      write (u, "(A)")  "*   Purpose: initialized a colored interaction"
      write (u, "(A)")

      write (u, "(A)")  "* Set up a process constants block"
      write (u, "(A)")
```

2122

```
      call os_data_init (os_data)
      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QCD"), var_str ("QCD.mdl"), &
           os_data, model)
      allocate (test_t :: core)

      associate (data => term%data)
        data%n_in = 2
        data%n_out = 3
        data%n_flv = 2
        data%n_hel = 2
        data%n_col = 2
        data%n_cin = 2

        allocate (data%flv_state (5, 2))
        data%flv_state (:,1) = [ 1, 21, 1, 21, 21]
        data%flv_state (:,2) = [ 2, 21, 2, 21, 21]

        allocate (data%hel_state (5, 2))
        data%hel_state (:,1) = [1, 1, 1, 1, 0]
        data%hel_state (:,2) = [1,-1, 1,-1, 0]

        allocate (data%col_state (2, 5, 2))
        data%col_state (:,:,1) = &
             reshape ([[1, 0], [2,-1], [3, 0], [2,-3], [0,0]], [2,5])
        data%col_state (:,:,2) = &
             reshape ([[1, 0], [2,-3], [3, 0], [2,-1], [0,0]], [2,5])

        allocate (data%ghost_flag (5, 2))
        data%ghost_flag(1:4,:) = .false.
        data%ghost_flag(5,:) = .true.

      end associate

      write (u, "(A)")  "* Set up the interaction"
      write (u, "(A)")

      call reset_interaction_counter ()
      call term%setup_interaction (core, model)
      call interaction_write (term%int, u)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_13"
    end subroutine processes_13
```

MD5 sums

Configure a process with structure functions (multi-channel) and compute MD5 sums

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_14, "processes_14", &
         "process configuration and MD5 sum", &
         u, results)
```
⟨*Processes: tests*⟩+≡
```
    subroutine processes_14 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(model_t), pointer :: model
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_channel_t), dimension(3) :: sf_channel

      write (u, "(A)")  "* Test output: processes_14"
      write (u, "(A)")  "*   Purpose: initialize a process with &
           &structure functions"
      write (u, "(A)")  "*            and compute MD5 sum"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a process object"
      write (u, "(A)")

      libname = "processes7"
      procname = libname
      run_id = "run7"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call lib%compute_md5sum ()

      call syntax_model_file_init ()

      allocate (process)
      call process%init &
           (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

      allocate (test_t :: core_template)
      allocate (phs_test_config_t :: phs_config_template)
      call process%init_component &
           (1, core_template, mci_template, phs_config_template)
```

2124

```
  write (u, "(A)")  "* Set beam, structure functions, and mappings"
  write (u, "(A)")

  sqrts = 1000
  call process%setup_beams_sqrts (sqrts)
  call process%configure_phs ()

  model => model_list%get_model_ptr (var_str ("Test"))
  pdg_in = 25
  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  call process%beam_config%allocate_sf_channels (3)

  allocate (sf_config (2))
  call sf_config(1)%init ([1], data)
  call sf_config(2)%init ([2], data)
  call process%init_sf_chain (sf_config)
  deallocate (sf_config)

  call sf_channel(1)%init (2)
  call process%set_sf_channel (1, sf_channel(1))

  call sf_channel(2)%init (2)
  call sf_channel(2)%activate_mapping ([1,2])
  call process%set_sf_channel (2, sf_channel(2))

  call sf_channel(3)%init (2)
  call sf_channel(3)%set_s_mapping ([1,2])
  call process%set_sf_channel (3, sf_channel(3))

  call process%setup_mci ()

  call process%compute_md5sum ()

  call process%write (.false., u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call process%final ()
  deallocate (process)

  call model_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: processes_14"

end subroutine processes_14
```

2125

Decay Process Evaluation

Initialize an evaluate a decay process.

⟨*Processes: execute tests*⟩+≡
```
    call test (processes_15, "processes_15", &
        "decay process", &
        u, results)
```

⟨*Processes: tests*⟩+≡
```
    subroutine processes_15 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(var_list_t), pointer :: var_list
      type(process_t), allocatable, target :: process
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      type(process_instance_t), allocatable, target :: process_instance
      type(particle_set_t) :: pset

      write (u, "(A)")  "* Test output: processes_15"
      write (u, "(A)")  "*   Purpose: initialize a decay process object"
      write (u, "(A)")

      write (u, "(A)")  "* Build and load a test library with one process"
      write (u, "(A)")

      libname = "processes15"
      procname = libname
      run_id = "run15"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call syntax_model_file_init ()

      call prc_test_create_library (libname, lib, scattering = .false., &
          decay = .true.)
      call model_list%read_model (var_str ("Test"), &
          var_str ("Test.mdl"), os_data, model)
      var_list => model_get_var_list_ptr (model)
      call var_list_set_real (var_list, &
          var_str ("ff"), 0.4_default, is_known = .true.)
      call model_parameters_update (model)

      write (u, "(A)")  "* Initialize a process object"
      write (u, "(A)")

      allocate (process)
```

2126

```
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

call process%setup_beams_decay ()
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

call reset_interaction_counter (3)

allocate (process_instance)
call process_instance%init (process)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()

write (u, "(A)")  "* Evaluate matrix element and square"
write (u, "(A)")

call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()
call process_instance%evaluate_trace ()
```

```
        call process_instance%write (u)

        call process_instance%get_trace (pset, 1)
        call process_instance%final ()
        deallocate (process_instance)

        write (u, "(A)")
        write (u, "(A)")  "* Particle content:"
        write (u, "(A)")

        call write_separator (u)
        call particle_set_write (pset, u)
        call write_separator (u)

        write (u, "(A)")
        write (u, "(A)")  "* Recover process instance"
        write (u, "(A)")

        call reset_interaction_counter (3)

        allocate (process_instance)
        call process_instance%init (process)
        call process_instance%choose_mci (1)
        call process_instance%set_trace (pset, 1)
        call process_instance%recover (1, 1, .true.)
        call process_instance%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call particle_set_final (pset)
        call process_instance%final ()
        deallocate (process_instance)

        call process%final ()
        deallocate (process)

        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: processes_15"

      end subroutine processes_15
```

Integration: decay

Activate the MC integrator for the decay object and use it to integrate over phase space.

⟨*Processes: execute tests*⟩+≡
```
      call test (processes_16, "processes_16", &
          "decay integration", &
```

```
          u, results)
```
⟨*Processes: tests*⟩+≡
```
     subroutine processes_16 (u)
       integer, intent(in) :: u
       type(process_library_t), target :: lib
       type(string_t) :: libname
       type(string_t) :: procname
       type(string_t) :: run_id
       type(os_data_t) :: os_data
       type(qcd_t) :: qcd
       class(rng_factory_t), allocatable :: rng_factory
       type(model_list_t) :: model_list
       type(model_t), pointer :: model
       type(var_list_t), pointer :: var_list
       type(process_t), allocatable, target :: process
       class(prc_core_t), allocatable :: core_template
       class(mci_t), allocatable :: mci_template
       class(phs_config_t), allocatable :: phs_config_template
       type(process_instance_t), allocatable, target :: process_instance

       write (u, "(A)")  "* Test output: processes_16"
       write (u, "(A)")  "*   Purpose: integrate a process without &
            &structure functions"
       write (u, "(A)")  "*            in a multi-channel configuration"
       write (u, "(A)")

       write (u, "(A)")  "* Build and initialize a process object"
       write (u, "(A)")

       libname = "processes16"
       procname = libname
       run_id = "run16"
       call os_data_init (os_data)
       allocate (rng_test_factory_t :: rng_factory)
       call syntax_model_file_init ()
       call prc_test_create_library (libname, lib, scattering = .false., &
            decay = .true.)
       call model_list%read_model (var_str ("Test"), &
            var_str ("Test.mdl"), os_data, model)
       var_list => model_get_var_list_ptr (model)
       call var_list_set_real (var_list, &
            var_str ("ff"), 0.4_default, is_known = .true.)
       call model_parameters_update (model)

       call reset_interaction_counter ()

       allocate (process)
       call process%init &
            (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

       allocate (test_t :: core_template)
       allocate (mci_midpoint_t :: mci_template)
       allocate (phs_single_config_t :: phs_config_template)
       call process%init_component &
```

```
         (1, core_template, mci_template, phs_config_template)

    write (u, "(A)")  "* Prepare a trivial beam setup"
    write (u, "(A)")

    call process%setup_beams_decay ()
    call process%configure_phs ()

    call process%setup_mci ()

    write (u, "(A)")  "* Complete process initialization"
    write (u, "(A)")

    call process%setup_terms ()
    call process%write (.false., u)

    write (u, "(A)")
    write (u, "(A)")  "* Create a process instance"
    write (u, "(A)")

    allocate (process_instance)
    call process_instance%init (process)

    write (u, "(A)")  "* Integrate with default test parameters"
    write (u, "(A)")

    call process%integrate (process_instance, 1, n_it=1, n_calls=10000)
    call process%final_integration (1)

    call process%write (.false., u)

    write (u, "(A)")
    write (u, "(A,ES13.7)")  " Integral divided by phs factor = ", &
         process%get_integral (1) &
         / process_instance%component(1)%k_seed%phs_factor

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_16"

  end subroutine processes_16
```

Decay Process Evaluation

Initialize an evaluate a decay process for a moving particle.

⟨*Processes: execute tests*⟩+≡

```
call test (processes_17, "processes_17", &
     "decay of moving particle", &
     u, results)
```

⟨*Processes: tests*⟩+≡

```
subroutine processes_17 (u)
  integer, intent(in) :: u
  type(process_library_t), target :: lib
  type(string_t) :: libname
  type(string_t) :: procname
  type(string_t) :: run_id
  type(os_data_t) :: os_data
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(var_list_t), pointer :: var_list
  type(process_t), allocatable, target :: process
  class(prc_core_t), allocatable :: core_template
  class(mci_t), allocatable :: mci_template
  class(phs_config_t), allocatable :: phs_config_template
  type(process_instance_t), allocatable, target :: process_instance
  type(particle_set_t) :: pset
  type(flavor_t) :: flv_beam
  real(default) :: m, p, E

  write (u, "(A)")  "* Test output: processes_17"
  write (u, "(A)")  "*   Purpose: initialize a decay process object"
  write (u, "(A)")

  write (u, "(A)")  "* Build and load a test library with one process"
  write (u, "(A)")

  libname = "processes17"
  procname = libname
  run_id = "run17"
  call os_data_init (os_data)
  allocate (rng_test_factory_t :: rng_factory)
  call syntax_model_file_init ()

  call prc_test_create_library (libname, lib, scattering = .false., &
       decay = .true.)
  call model_list%read_model (var_str ("Test"), &
       var_str ("Test.mdl"), os_data, model)
  var_list => model_get_var_list_ptr (model)
  call var_list_set_real (var_list, &
       var_str ("ff"), 0.4_default, is_known = .true.)
  call model_parameters_update (model)

  write (u, "(A)")  "* Initialize a process object"
  write (u, "(A)")
```

2131

```
allocate (process)
call process%init &
     (procname, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

call process%setup_beams_decay (rest_frame = .false.)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

call reset_interaction_counter (3)

allocate (process_instance)
call process_instance%init (process)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set parent momentum and random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])

call flavor_init (flv_beam, 25, model)
m = flavor_get_mass (flv_beam)
p = 3 * m / 4
E = sqrt (m**2 + p**2)
call process_instance%set_beam_momenta ([vector4_moving (E, p, 3)])

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
```

```
      call process_instance%compute_hard_kinematics ()

      write (u, "(A)")  "* Evaluate matrix element and square"
      write (u, "(A)")

      call process_instance%compute_eff_kinematics ()
      call process_instance%evaluate_expressions ()
      call process_instance%compute_other_channels ()
      call process_instance%evaluate_trace ()
      call process_instance%write (u)

      call process_instance%get_trace (pset, 1)
      call process_instance%final ()
      deallocate (process_instance)

      write (u, "(A)")
      write (u, "(A)")  "* Particle content:"
      write (u, "(A)")

      call write_separator (u)
      call particle_set_write (pset, u)
      call write_separator (u)

      write (u, "(A)")
      write (u, "(A)")  "* Recover process instance"
      write (u, "(A)")

      call reset_interaction_counter (3)

      allocate (process_instance)
      call process_instance%init (process)

      call process_instance%choose_mci (1)
      call process_instance%set_trace (pset, 1)
      call process_instance%recover (1, 1, .true.)
      call process_instance%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call particle_set_final (pset)
      call process_instance%final ()
      deallocate (process_instance)

      call process%final ()
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: processes_17"

    end subroutine processes_17
```

15.6 Process Stacks

For storing and handling multiple processes, we define process stacks. These are ordinary stacks where new process entries are pushed onto the top. We allow for multiple entries with identical process ID, but distinct run ID.

The implementation is essentially identical to the `prclib_stacks` module above. Unfortunately, Fortran supports no generic programming, so we do not make use of this fact.

When searching for a specific process ID, we will get (a pointer to) the top-most process entry with that ID on the stack, which was entered last. Usually, this is the best version of the process (in terms of integral, etc.) Thus the stack terminology makes sense.

⟨`process_stacks.f90`⟩≡
  ⟨*File header*⟩

  ```
  module process_stacks
  ```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use unit_tests
    use os_interface
    use sm_qcd
    use variables
    use models
    use rng_base

    use process_libraries
    use prc_test
    use processes
  ```

  ⟨*Standard module head*⟩

  ⟨*Process stacks: public*⟩

  ⟨*Process stacks: types*⟩

  ```
  contains
  ```

  ⟨*Process stacks: procedures*⟩

  ⟨*Process stacks: tests*⟩

  ```
  end module process_stacks
  ```

15.6.1 The process entry type

A process entry is a process object, augmented by a pointer to the next entry. We do not need specific methods, all relevant methods are inherited.

On higher level, processes should be prepared as process entry objects.

⟨Process stacks: public⟩≡
```
    public :: process_entry_t
```

⟨Process stacks: types⟩≡
```
    type, extends (process_t) :: process_entry_t
       type(process_entry_t), pointer :: next => null ()
    end type process_entry_t
```

15.6.2   The process stack type

For easy conversion and lookup it is useful to store the filling number in the object. The content is stored as a linked list.

The `var_list` component stores process-specific results, so they can be retrieved as (pseudo) variables.

⟨Process stacks: public⟩+≡
```
    public :: process_stack_t
```

⟨Process stacks: types⟩+≡
```
    type :: process_stack_t
       integer :: n = 0
       type(process_entry_t), pointer :: first => null ()
       type(var_list_t), pointer :: var_list => null ()
     contains
     ⟨Process stacks: process stack: TBP⟩
    end type process_stack_t
```

Finalizer. Iteratively deallocate the stack entries. The resulting empty stack can be immediately recycled, if necessary.

⟨Process stacks: process stack: TBP⟩≡
```
    procedure :: final => process_stack_final
```

⟨Process stacks: procedures⟩≡
```
    subroutine process_stack_final (object)
      class(process_stack_t), intent(inout) :: object
      type(process_entry_t), pointer :: process
      if (associated (object%var_list)) then
         call var_list_final (object%var_list)
         deallocate (object%var_list)
      end if
      do while (associated (object%first))
         process => object%first
         object%first => process%next
         call process%final ()
         deallocate (process)
      end do
      object%n = 0
    end subroutine process_stack_final
```

Output. The processes on the stack will be ordered LIFO, i.e., backwards.

⟨Process stacks: process stack: TBP⟩+≡
```
    procedure :: write => process_stack_write
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_write (object, unit, pacify)
  class(process_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  type(process_entry_t), pointer :: process
  integer :: u
  u = output_unit (unit)
  call write_separator_double (u)
  select case (object%n)
  case (0)
     write (u, "(1x,A)")  "Process stack: [empty]"
     call write_separator_double (u)
  case default
     write (u, "(1x,A)")  "Process stack:"
     process => object%first
     do while (associated (process))
        call process%write (.false., u, pacify = pacify)
        process => process%next
     end do
  end select
end subroutine process_stack_write
```

The variable list is printed by a separate routine, since it should be linked to the global variable list, anyway.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: write_var_list => process_stack_write_var_list
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_write_var_list (object, unit)
  class(process_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  if (associated (object%var_list)) then
     call var_list_write (object%var_list, unit)
  end if
end subroutine process_stack_write_var_list
```

Short output.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: show => process_stack_show
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_show (object, unit)
  class(process_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(process_entry_t), pointer :: process
  integer :: u
  u = output_unit (unit)
  select case (object%n)
  case (0)
  case default
     process => object%first
     do while (associated (process))
        call process%show (u, verbose=.false.)
```

```
          process => process%next
        end do
      end select
    end subroutine process_stack_show
```

15.6.3   Push

We take a process pointer and push it onto the stack. The previous pointer is nullified. Subsequently, the process is 'owned' by the stack and will be finalized when the stack is deleted.

⟨*Process stacks: process stack: TBP*⟩+≡
```
    procedure :: push => process_stack_push
```

⟨*Process stacks: procedures*⟩+≡
```
    subroutine process_stack_push (stack, process)
      class(process_stack_t), intent(inout) :: stack
      type(process_entry_t), intent(inout), pointer :: process
      process%next => stack%first
      stack%first => process
      process => null ()
      stack%n = stack%n + 1
    end subroutine process_stack_push
```

Initialize the process variable list and link it to a global variable list.

⟨*Process stacks: process stack: TBP*⟩+≡
```
    procedure :: init_var_list => process_stack_init_var_list
```

⟨*Process stacks: procedures*⟩+≡
```
    subroutine process_stack_init_var_list (stack, var_list)
      class(process_stack_t), intent(inout) :: stack
      type(var_list_t), intent(inout), optional :: var_list
      allocate (stack%var_list)
      if (present (var_list))  call var_list_link (var_list, stack%var_list)
    end subroutine process_stack_init_var_list
```

Initialize process variables for a given process ID, without setting values.

⟨*Process stacks: process stack: TBP*⟩+≡
```
    procedure :: init_result_vars => process_stack_init_result_vars
```

⟨*Process stacks: procedures*⟩+≡
```
    subroutine process_stack_init_result_vars (stack, id)
      class(process_stack_t), intent(inout) :: stack
      type(string_t), intent(in) :: id
      call var_list_init_num_id (stack%var_list, id)
      call var_list_init_process_results (stack%var_list, id)
    end subroutine process_stack_init_result_vars
```

Fill process variables with values. This is executed after the integration pass.

  Note: We set only integral and error. With multiple MCI records possible, the results for `n_calls`, `chi2` etc. are not necessarily unique. (We might set the efficiency, though.)

⟨*Process stacks: process stack: TBP*⟩+≡

```
      procedure :: fill_result_vars => process_stack_fill_result_vars
```

⟨*Process stacks: procedures*⟩+≡
```
    subroutine process_stack_fill_result_vars (stack, id)
      class(process_stack_t), intent(inout) :: stack
      type(string_t), intent(in) :: id
      type(process_t), pointer :: process
      process => stack%get_process_ptr (id)
      if (associated (process)) then
         call var_list_init_num_id (stack%var_list, id, process%get_num_id ())
         if (process%has_integral ()) then
            call var_list_init_process_results (stack%var_list, id, &
                 integral = process%get_integral (), &
                 error = process%get_error ())
         end if
      else
         call msg_bug ("process_stack_fill_result_vars: unknown process ID")
      end if
    end subroutine process_stack_fill_result_vars
```

15.6.4 Data Access

Tell if a process exists.

⟨*Process stacks: process stack: TBP*⟩+≡
```
    procedure :: exists => process_stack_exists
```

⟨*Process stacks: procedures*⟩+≡
```
    function process_stack_exists (stack, id) result (flag)
      class(process_stack_t), intent(in) :: stack
      type(string_t), intent(in) :: id
      logical :: flag
      type(process_t), pointer :: process
      process => stack%get_process_ptr (id)
      flag = associated (process)
    end function process_stack_exists
```

Return a pointer to a process with specific ID.

⟨*Process stacks: process stack: TBP*⟩+≡
```
    procedure :: get_process_ptr => process_stack_get_process_ptr
```

⟨*Process stacks: procedures*⟩+≡
```
    function process_stack_get_process_ptr (stack, id) result (ptr)
      class(process_stack_t), intent(in) :: stack
      type(string_t), intent(in) :: id
      type(process_t), pointer :: ptr
      type(process_entry_t), pointer :: entry
      ptr => null ()
      entry => stack%first
      do while (associated (entry))
         if (entry%get_id () == id) then
            ptr => entry%process_t
            return
         end if
```

```
        entry => entry%next
      end do
    end function process_stack_get_process_ptr
```

15.6.5   Unit tests

⟨*Process stacks: public*⟩+≡
```
    public :: process_stacks_test
```

⟨*Process stacks: tests*⟩≡
```
    subroutine process_stacks_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Process stacks: execute tests*⟩
```
    end subroutine process_stacks_test
```

Write an empty process stack

The most trivial test is to write an uninitialized process stack.

⟨*Process stacks: execute tests*⟩≡
```
    call test (process_stacks_1, "process_stacks_1", &
        "write an empty process stack", &
        u, results)
```

⟨*Process stacks: tests*⟩+≡
```
    subroutine process_stacks_1 (u)
      integer, intent(in) :: u
      type(process_stack_t) :: stack

      write (u, "(A)")  "* Test output: process_stacks_1"
      write (u, "(A)")  "*   Purpose: display an empty process stack"
      write (u, "(A)")

      call stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_stacks_1"

    end subroutine process_stacks_1
```

Fill a process stack

Fill a process stack with two (identical) processes.

⟨*Process stacks: execute tests*⟩+≡
```
    call test (process_stacks_2, "process_stacks_2", &
        "fill a process stack", &
        u, results)
```

⟨*Process stacks: tests*⟩+≡

```fortran
    subroutine process_stacks_2 (u)
      integer, intent(in) :: u
      type(process_stack_t) :: stack
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(process_entry_t), pointer :: process => null ()

      write (u, "(A)")  "* Test output: process_stacks_2"
      write (u, "(A)")  "*   Purpose: fill a process stack"
      write (u, "(A)")

      write (u, "(A)")  "* Build, initialize and store two test processes"
      write (u, "(A)")

      libname = "process_stacks2"
      procname = libname
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)
      call syntax_model_file_init ()

      allocate (process)
      run_id = "run1"
      call process%init &
           (procname, run_id, lib, os_data, qcd, rng_factory, model_list)
      call stack%push (process)

      allocate (process)
      run_id = "run2"
      call process%init &
           (procname, run_id, lib, os_data, qcd, rng_factory, model_list)
      call stack%push (process)

      call stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call stack%final ()

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_stacks_2"

    end subroutine process_stacks_2
```

2140

Fill a process stack

Fill a process stack with two (identical) processes.

⟨Process stacks: execute tests⟩+≡
```
    call test (process_stacks_3, "process_stacks_3", &
         "process variables", &
         u, results)
```

⟨Process stacks: tests⟩+≡
```
    subroutine process_stacks_3 (u)
      integer, intent(in) :: u
      type(process_stack_t) :: stack
      type(model_list_t) :: model_list
      type(string_t) :: procname
      type(process_entry_t), pointer :: process => null ()
      type(process_instance_t), target :: process_instance

      write (u, "(A)")  "* Test output: process_stacks_3"
      write (u, "(A)")  "*   Purpose: setup process variables"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process variables"
      write (u, "(A)")

      procname = "processes_test"
      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize process variables"
      write (u, "(A)")

      call stack%init_var_list ()
      call stack%init_result_vars (procname)
      call stack%write_var_list (u)

      write (u, "(A)")
      write (u, "(A)")  "* Build and integrate a test process"
      write (u, "(A)")

      allocate (process)
      call prepare_test_process (process%process_t, process_instance, model_list)
      call process%integrate (process_instance, 1, 1, 1000)
      call process_instance%final ()
      call process%final_integration (1)
      call stack%push (process)

      write (u, "(A)")  "* Fill process variables"
      write (u, "(A)")

      call stack%fill_result_vars (procname)
      call stack%write_var_list (u)

      write (u, "(A)")
```

2141

```
    write (u, "(A)")  "* Cleanup"

    call stack%final ()

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_stacks_3"

  end subroutine process_stacks_3
```

Chapter 16

Parton shower and interface to hadronization

This is the code for the WHIZARD QCD parton shower for final state radiation (FSR) and initial state radiation (ISR) as well as the interface to the PYTHIA module for showering and hadronization.

⟨*Version*⟩≡
```
  2.2.0
```

⟨*Date*⟩≡
```
  Mar 03 2014
```

⟨*File header*⟩≡
```
  ! WHIZARD ⟨Version⟩ ⟨Date⟩
  !
  ! Copyright (C) 1999-2014 by
  !     Wolfgang Kilian <kilian@physik.uni-siegen.de>
  !     Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
  !     Juergen Reuter <juergen.reuter@desy.de>
  !
  !     with contributions from
  !     Christian Speckner <cnspeckn@googlemail.com>
  !     and Fabian Bach, Felix Braam, Sebastian Schmidt, Daniel Wiesler
  !
  ! WHIZARD is free software; you can redistribute it and/or modify it
  ! under the terms of the GNU General Public License as published by
  ! the Free Software Foundation; either version 2, or (at your option)
  ! any later version.
  !
  ! WHIZARD is distributed in the hope that it will be useful, but
  ! WITHOUT ANY WARRANTY; without even the implied warranty of
  ! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  ! GNU General Public License for more details.
  !
  ! You should have received a copy of the GNU General Public License
  ! along with this program; if not, write to the Free Software
  ! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
  !
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! This file has been stripped of most comments.  For documentation, refer
! to the source 'shower.nw'
```

We are strict with our names:

⟨*Standard module head*⟩≡
```
    implicit none
    private
```

This is the way to envoke the kinds module (not contained in this source)

⟨*Use kinds*⟩≡
```
    use kinds, only: default !NODEP!
```

And we make heavy use of variable-length strings

⟨*Use strings*⟩≡
```
    use iso_varying_string, string_t => varying_string !NODEP!
```

16.1   Basics of the shower

⟨`shower_base.f90`⟩≡
  ⟨*File header*⟩
```

  module shower_base
```
  ⟨*Use kinds*⟩
```
    use constants !NODEP!
    use tao_random_numbers !NODEP!
```
  ⟨*Standard module head*⟩

  ⟨*Shower base: public*⟩

  ⟨*Shower base: parameters*⟩

  ⟨*Shower base: interfaces*⟩
```
  contains
```
  ⟨*Shower base: procedures*⟩
```
  end module shower_base
```

### 16.1.1   PDF Interface of the shower

⟨*Shower base: public*⟩≡
```
    public :: shower_pdf
```

⟨*Shower base: interfaces*⟩≡
```
    interface
      subroutine shower_pdf (set, x, q, ff)
        integer, intent(in) :: set
        double precision, intent(in) :: x, q
        double precision, dimension(-6:6), intent(out) :: ff
      end subroutine shower_pdf
```

```
    end interface
```

16.1.2 Parameters

This decides whether to print out additional information.

⟨*Shower base: parameters*⟩≡
```
    logical, parameter, public :: D_print = .false.
```

These parameters are the cut-off scale t_{cut}, given in GeV^2 (`PARJ(82)`), the cut-off scale for the P_t^2-ordered shower in GeV^2, and the two shower parameters `PARP(72)` and `PARP(61)`, respectively.

⟨*Shower base: parameters*⟩+≡
```
    real(default), public :: D_Min_t = one
    real(default), public :: D_min_scale = 0.5_default
    real(default), public :: D_Lambda_fsr = 0.29_default
    real(default), public :: D_Lambda_isr = 0.29_default
```

The parameter `MSTJ(45)` gives the maximum number of flavours in gluon decay to quarks:

⟨*Shower base: parameters*⟩+≡
```
    integer, public :: D_Nf = 5
```

These two parameters decide whether to use constant or running $alpha_s$ in the form of the function $D\_alpha_s(t)$ for the FSR and ISR (`MSTJ(44)`, `MSTP(64)`), respectively. The next is the parameter `PARU(111)`, which sets the value for constant α_s, and the flag whether to use P_t-ordered ISR.

⟨*Shower base: parameters*⟩+≡
```
    logical, public :: D_running_alpha_s_fsr = .true.
    logical, public :: D_running_alpha_s_isr = .true.
    real(default), public :: D_constalpha_s = 0.20_default
    logical, public :: isr_pt_ordered = .false.
```

This flag sets emitted timelike partons in the spacelike shower on shell, true corresponds to `MSTP(63)` = 0.

⟨*Shower base: parameters*⟩+≡
```
    logical, public :: isr_only_onshell_emitted_partons = .true.
```

Flag whether ISR is angular ordered (`MSTP(62)`).

⟨*Shower base: parameters*⟩+≡
```
    logical, public :: isr_angular_ordered = .true.
```

Treating either u and d, or all quarks except t as massless:

⟨*Shower base: parameters*⟩+≡
```
    logical, public :: treat_light_quarks_massless = .true.
    logical, public :: treat_duscb_quarks_massless = .false.
```

The width and the cutoff of the Gaussian primordial k_t distribution, `PARP(91)` and `PARP(93)`, in GeV:

⟨*Shower base: parameters*⟩+≡
```
    real(default), public :: primordial_kt_width = 1.5_default
    real(default), public :: primordial_kt_cutoff = five
```

The parameter `PARP(66)`, the minimal energy of the emitted timelike parton in the ISR (`PARP(65)`), the factor for the first scale, as well as the factor by which the integral in the Sudakhov factor is suppressed for the respective first scale in ISR. Higher values here correspond to higher starting scales.

⟨*Shower base: parameters*⟩+≡
```
real(default), public :: maxz_isr = 0.999_default
real(default), public :: minenergy_timelike = one
real(default), public :: tscalefactor_isr = one
real(default), public :: first_integral_suppression_factor = one
```

Temporary parameters for the P_t-ordered shower:

⟨*Shower base: parameters*⟩+≡
```
real(default), public :: scalefactor1 = 0.02_default
real(default), public :: scalefactor2 = 0.02_default
```

Then, there are the variables for the PDF functions:

⟨*Shower base: parameters*⟩+≡
```
integer, public :: shower_pdf_set = 0
```

And the PDF function itself, as a procedure pointer:

⟨*Shower base: parameters*⟩+≡
```
procedure(shower_pdf), pointer, public :: shower_pdf_func
```

⟨*Shower base: procedures*⟩≡
```
subroutine randomseed (seed)
  integer, intent(in), optional :: seed
  integer :: clock

  if (present (seed)) then
     clock = seed
  else
     call system_clock (count = clock)
  end if
  call tao_random_seed (clock)
end subroutine randomseed
```

⟨*Shower base: public*⟩+≡
```
public :: D_alpha_s_isr
public :: D_alpha_s_fsr
```

⟨*Shower base: procedures*⟩+≡
```
function D_alpha_s_isr (tin) result(alpha_s)
  real(default), intent(in) :: tin
  real(default) :: b, t
  real(default) :: alpha_s

!    arbitrary lower cut off for scale
!    t = MAX(max(one * D_Min_t, 1.1_default * D_Lambda_isr**2), ABS(tin))
  t = max (max (0.1_default * D_Min_t, &
        1.1_default * D_Lambda_isr**2), abs(tin))

  if (D_running_alpha_s_isr) then
     b = (33._default - two * number_of_flavors(t)) / (12._default * pi)
     alpha_s = one / (b * log(t / (D_Lambda_isr**2)))
```

2146

```
      else
         alpha_s = D_constalpha_s
      end if
   end function D_alpha_s_isr

   function D_alpha_s_fsr (tin) result(alpha_s)
     real(default), intent(in) :: tin
     real(default) :: b, t
     real(default) :: alpha_s

 !   arbitrary lower cut off for scale
 !   t= max( max (one * D_Min_t, 1.1_default * D_Lambda**2), ABS(tin))
     t = max (max (0.1_default * D_Min_t, &
          1.1_default * D_Lambda_fsr**2), abs(tin))

     if (D_running_alpha_s_fsr) then
        b = (33._default - two * number_of_flavors(t)) / (12._default * pi)
        alpha_s = one / (b * log(t / (D_Lambda_fsr**2)))
     else
        alpha_s = D_constalpha_s
     end if
   end function D_alpha_s_fsr
```

Mass and mass squared selection functions. All masses are in GeV. Light quarks are assumed to be ordered, $m_1 < m_2 < m_3 \ldots$, and they get current masses, not elementary ones. Mesons and baryons other than proton and neutron are needed as beam-remnants. Particles with PDG number zero are taken massless, as well as proper beam remnants and any other particles.

⟨*Shower base: public*⟩+≡
```
    public :: mass_type
    public :: mass_squared_type
```

⟨*Shower base: procedures*⟩+≡
```
    function mass_type (type) result (mass)
      integer, intent(in) :: type
      real(default) :: mass
      mass = sqrt (mass_squared_type (type))
    end function mass_type

    function mass_squared_type (type) result (mass2)
      integer, intent(in) :: type
      real(default) :: mass2

      select case (abs (type))
      case (1,2)
         if (treat_light_quarks_massless .or. &
             treat_duscb_quarks_massless) then
           mass2 = zero
         else
           mass2 = 0.330_default**2
         end if
      case (3)
         if (treat_duscb_quarks_massless) then
            mass2 = zero
```

2147

```fortran
      else
         mass2 = 0.500_default**2
      end if
   case (4)
      if (treat_duscb_quarks_massless) then
         mass2 = zero
      else
         mass2 = 1.500_default**2
      end if
   case (5)
      if (treat_duscb_quarks_massless) then
         mass2 = zero
      else
         mass2 = 4.800_default**2
      end if
   case (6)
      mass2 = 175.00_default**2
   case (21)
      mass2 = zero
   case (2112)   ! Neutron
      mass2 = 0.939565_default**2
   case (2212)   ! Proton
      mass2 = 0.93827_default**2
   case (411) ! D+
      mass2 = 1.86960_default**2
   case (421) ! D0
      mass2 = 1.86483_default**2
   case (511) ! B0
      mass2 = 5.27950_default**2
   case (521) ! B+
      mass2 = 5.27917_default**2
   case (2224) !Delta++
      mass2 = 1.232_default**2
   case (3212) !Sigma0
      mass2 = 1.192642_default**2
   case (3222) !Sigma+
      mass2 = 1.18937_default**2
   case (4212) ! Sigma_c+
      mass2 = 2.4529_default**2
   case (4222) ! Sigma_c++
      mass2 = 2.45402_default**2
   case (5212) ! Sigma_b0
      mass2 = 5.8152_default**2
   case (5222) ! Sigma_b+
      mass2 = 5.8078_default**2
   case (0) ! I take 0 to be partons whose type is not yet clear
      mass2 = zero
   case (9999) ! beam remnant
      mass2 = zero ! don't know how to handle the beamremnant
   case default !others not implemented
      mass2 = zero
   end select
end function mass_squared_type
```

The number of flavours allowed in an actual $g \to qq$ splitting.

⟨*Shower base: public*⟩+≡
```
    public :: number_of_flavors
```

⟨*Shower base: procedures*⟩+≡
```
    function number_of_flavors (t) result (nr)
      real(default), intent(in) :: t
      integer :: nr
      integer :: i
      nr = 0
      if (t < 0.25_default * D_Min_t) return   ! arbitrary cut off ?WRONG?
      do i = 1, min (D_Nf, 3)
         ! to do: take heavier quarks(-> cuts on allowed costheta in g->qq) into account
         if ((four * mass_squared_type (i) + D_Min_t) < t ) then
            nr = i
         else
            exit
         end if
      end do
    end function number_of_flavors
```

Splitting functions. They are also defined in `sm_physics`. This should be unified at some point.

⟨*Shower base: public*⟩+≡
```
    public :: P_qqg
    public :: P_gqq
    public :: P_ggg
```

⟨*Shower base: procedures*⟩+≡
```
    function P_qqg (z) result (P)
      !!! quark => quark + gluon
      real(default), intent(in) :: z
      real(default) :: P
      P = (four / three) * (one + z**2) / (one - z)
    end function P_qqg

    function P_gqq (z) result (P)
      !!! gluon => quark + antiquark
      real(default), intent(in) :: z
      real(default) :: P
      P = 0.5_default * (z**2 + (one - z)**2)
      !!! anti-symmetrized version -> needs change of first and second daughter
      !!! in 50% of branchings
      !     P = (one - z)**2
    end function P_gqq

    function P_ggg (z) result (P)
      !!! gluon => gluon + gluon
      real(default), intent(in) :: z
      real(default) :: P
      P = three * ((one - z) / z + z / (one - z) + z * (one - z))
      !!! anti-symmetrized version -> needs to by symmetrized in color connections
      !     P = three * ( two * z / (one - z) + z * (one - z) )
    end function P_ggg
```

Analytically integrated splitting kernels.

⟨*Shower base: public*⟩+≡
```
    public :: integral_over_P_gqq
    public :: integral_over_P_ggg
    public :: integral_over_P_qqg
```

⟨*Shower base: procedures*⟩+≡
```
    function integral_over_P_gqq (zmin, zmax) result (integral)
      real(default), intent(in) :: zmin, zmax
      real(default) :: integral
      integral = 0.5_default * ((two / three) * &
           (zmax**3 - zmin**3) - (zmax**2 - zmin**2) + (zmax - zmin))
    end function integral_over_P_gqq

    function integral_over_P_ggg (zmin, zmax) result (integral)
      real(default), intent(in) :: zmin, zmax
      real(default) :: integral
      integral = three * ((log(zmax) - two * zmax - &
           log(one - zmax) + zmax**2 / two - zmax**3 / three) - &
           (log(zmin) - zmin - zmin - log(one - zmin) + zmin**2 &
           / two - zmin**3 / three) )
    end function integral_over_P_ggg

    function integral_over_P_qqg (zmin, zmax) result (integral)
      real(default), intent(in) :: zmin, zmax
      real(default) :: integral
      integral = (two / three) * ( - zmax**2 + zmin**2 - &
           two * (zmax - zmin) + four * &
           log((one - zmin) / (one - zmax)))
    end function integral_over_P_qqg
```

Methods to set parameters, once there should be a better interface.

⟨*Shower base: public*⟩+≡
```
    public :: shower_set_D_min_t
    public :: shower_set_D_lambda_fsr
    public :: shower_set_D_lambda_isr
    public :: shower_set_D_Nf
    public :: shower_set_D_running_alpha_s_fsr
    public :: shower_set_D_running_alpha_s_isr
    public :: shower_set_D_constantalpha_s
    public :: shower_set_isr_pt_ordered
    public :: shower_set_isr_angular_ordered
    public :: shower_set_primordial_kt_width
    public :: shower_set_primordial_kt_cutoff
    public :: shower_set_maxz_isr
    public :: shower_set_minenergy_timelike
    public :: shower_set_tscalefactor_isr
    public :: shower_set_isr_only_onshell_emitted_partons
    public :: shower_set_pdf_func
    public :: shower_set_pdf_set
```

⟨*Shower base: procedures*⟩+≡
```
    subroutine shower_set_D_Min_t (input)
      real(default) :: input
```

```
    D_Min_t = input
end subroutine shower_set_D_Min_t

subroutine shower_set_D_Lambda_fsr (input)
  real(default) :: input
  D_Lambda_fsr = input
end subroutine shower_set_D_Lambda_fsr

subroutine shower_set_D_Lambda_isr (input)
  real(default) :: input
  D_Lambda_isr = input
end subroutine shower_set_D_Lambda_isr

subroutine shower_set_D_Nf (input)
  integer :: input
  D_Nf = input
end subroutine shower_set_D_Nf

subroutine shower_set_D_running_alpha_s_fsr (input)
  logical :: input
  D_running_alpha_s_fsr = input
end subroutine shower_set_D_running_alpha_s_fsr

subroutine shower_set_D_running_alpha_s_isr (input)
  logical :: input
  D_running_alpha_s_isr = input
end subroutine shower_set_D_running_alpha_s_isr

subroutine shower_set_D_constantalpha_s (input)
  real(default) :: input
  D_constalpha_s = input
end subroutine shower_set_D_constantalpha_s

subroutine shower_set_isr_pt_ordered (input)
  logical :: input
  isr_pt_ordered = input
end subroutine shower_set_isr_pt_ordered

subroutine shower_set_isr_angular_ordered (input)
  logical :: input
  isr_angular_ordered = input
end subroutine shower_set_isr_angular_ordered

subroutine shower_set_primordial_kt_width (input)
  real(default) :: input
  primordial_kt_width = input
end subroutine shower_set_primordial_kt_width

subroutine shower_set_primordial_kt_cutoff (input)
  real(default) :: input
  primordial_kt_cutoff = input
end subroutine shower_set_primordial_kt_cutoff

subroutine shower_set_maxz_isr (input)
```

```
    real(default) :: input
    maxz_isr = input
  end subroutine shower_set_maxz_isr

  subroutine shower_set_minenergy_timelike (input)
    real(default) :: input
    minenergy_timelike = input
  end subroutine shower_set_minenergy_timelike

  subroutine shower_set_tscalefactor_isr (input)
    real(default) :: input
    tscalefactor_isr = input
  end subroutine shower_set_tscalefactor_isr

  subroutine shower_set_isr_only_onshell_emitted_partons (input)
    logical :: input
    isr_only_onshell_emitted_partons = input
  end subroutine shower_set_isr_only_onshell_emitted_partons

  subroutine shower_set_pdf_func (func)
    procedure(shower_pdf), pointer, intent(in) :: func
    shower_pdf_func => func
  end subroutine shower_set_pdf_func

  subroutine shower_set_pdf_set (set)
    integer, intent(in) :: set
    shower_pdf_set = set
  end subroutine shower_set_pdf_set
```

16.2   Parton module for the shower

⟨shower_partons.f90⟩≡
  ⟨*File header*⟩

```
  module shower_partons
```

  ⟨*Use kinds*⟩
```
    use constants !NODEP!
    use limits, only: TAB !NODEP!
    use file_utils !NODEP!
    use diagnostics !NODEP!
    use tao_random_numbers !NODEP!
    use lorentz !NODEP!
    use shower_base
```

  ⟨*Standard module head*⟩

  ⟨*Shower partons: public*⟩

  ⟨*Shower partons: types*⟩

```
  contains
```

⟨*Shower partons: procedures*⟩

```
end module shower_partons
```

16.2.1 The basic type defintions

The type `parton_t` defines a parton for the shower. The `x` value of the parton is only needed for spacelike showers. The pointer `initial` is only needed for partons in initial showers, it points to the hadron the parton is coming from. An auxiliary value for the P_t-ordered ISR is `aux_pt`. Then, there are two auxiliary entries for the clustering of CKKW pseudo weights and CKKW matching, `ckkwlabel` and `ckkwscale`.

⟨*Shower partons: public*⟩≡
```
    public :: parton_t
```
⟨*Shower partons: types*⟩≡
```
    type :: parton_t
       integer :: nr = 0
       integer :: type = 0
       type(vector4_t) :: momentum = vector4_null
       real(default) :: t  = zero
       real(default) :: scale = zero
       real(default) :: z = zero
       real(default) :: costheta = zero
       real(default) :: x = zero
       logical :: simulated = .false.
       logical :: belongstoFSR = .true.
       logical :: belongstointeraction = .false.
       type(parton_t), pointer :: parent => null ()
       type(parton_t), pointer :: child1 => null ()
       type(parton_t), pointer :: child2 => null ()
       type(parton_t), pointer :: initial => null ()
       integer :: c1 = 0, c2 = 0
       integer :: aux_pt = 0
       integer :: ckkwlabel = 0
       real(default) :: ckkwscale = zero
       integer :: ckkwtype = -1
       integer :: interactionnr = 0
    end type parton_t
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_pointer_t
```
⟨*Shower partons: types*⟩+≡
```
    type :: parton_pointer_t
       type(parton_t), pointer :: p => null ()
    end type parton_pointer_t
```

16.2.2 Routines

⟨*Shower partons: public*⟩+≡
```
    public :: parton_copy
```

⟨Shower partons: procedures⟩≡
```
    subroutine parton_copy (prt1, prt2)
      type(parton_t), intent(in) :: prt1
      type(parton_t), intent(out) :: prt2
      prt2%nr = prt1%nr
      prt2%type = prt1%type
      prt2%momentum = prt1%momentum
      prt2%t = prt1%t
      prt2%scale = prt1%scale
      prt2%z = prt1%z
      prt2%costheta = prt1%costheta
      prt2%x = prt1%x
      prt2%simulated = prt1%simulated
      prt2%belongstoFSR = prt1%belongstoFSR
      prt2%belongstointeraction = prt1%belongstointeraction
      prt2%interactionnr = prt1%interactionnr
      if (associated (prt1%parent))  prt2%parent  => prt1%parent
      if (associated (prt1%child1))  prt2%child1  => prt1%child1
      if (associated (prt1%child2))  prt2%child2  => prt1%child2
      if (associated (prt1%initial)) prt2%initial => prt1%initial
      prt2%c1 = prt1%c1
      prt2%c2 = prt1%c2
      prt2%aux_pt = prt1%aux_pt
    end subroutine parton_copy
```

This returns the angle between the daughters assuming them to be massless.

⟨Shower partons: public⟩+≡
```
    public :: parton_get_costheta
```

⟨Shower partons: procedures⟩+≡
```
    function parton_get_costheta (prt) result (costheta)
      type(parton_t), intent(in) :: prt
      real(default) :: costheta
      if (prt%z * (one - prt%z) * parton_get_energy(prt)**2 > zero) then
         costheta = one - prt%t / (two * prt%z * (one - prt%z) * &
               parton_get_energy(prt)**2)
      else
         costheta = - one
      end if
    end function parton_get_costheta
```

The same for massive daughters.

⟨Shower partons: public⟩+≡
```
    public :: parton_get_costheta_correct
```

⟨Shower partons: procedures⟩+≡
```
    function parton_get_costheta_correct (prt) result (costheta)
      type(parton_t), intent(in) :: prt
      real(default) :: costheta
      if (parton_is_branched (prt)) then
         if (parton_is_simulated (prt%child1) .and. &
               parton_is_simulated (prt%child2) .and. &
               sqrt (max (zero, (prt%z)**2 * parton_get_energy(prt)**2 &
                                     - prt%child1%t)) * &
```

```
                  sqrt (max (zero, (1.-prt%z)**2 * parton_get_energy(prt)**2 &
                                    - prt%child2%t)) > zero) then
                costheta = &
                    (prt%t - prt%child1%t - prt%child2%t - 2. * prt%z * &
                    (1.-prt%z) * parton_get_energy(prt)**2) / &
                    (-2.* sqrt((prt%z)**2 * parton_get_energy(prt)**2 &
                    - prt%child1%t) * &
                    sqrt( (1.-prt%z)**2 * parton_get_energy(prt)**2 - prt%child2%t))
            else
               costheta = parton_get_costheta (prt)
            end if
         else
            costheta = parton_get_costheta (prt)
         end if
      end function parton_get_costheta_correct
```

This function returns the angle between the momentum vectors of the parton
and first daughter.

⟨*Shower partons: public*⟩+≡
```
      public :: parton_get_costheta_motherfirst
```

⟨*Shower partons: procedures*⟩+≡
```
      function parton_get_costheta_motherfirst (prt) result (costheta)
        type(parton_t), intent(in) :: prt
        real(default) :: costheta
        if (parton_is_branched (prt)) then
           if ((parton_is_simulated (prt%child1) .or. parton_is_final (prt%child1) &
                .or. parton_is_branched (prt%child1)) .and. &
                (parton_is_simulated (prt%child2) .or. &
                parton_is_final (prt%child2) &
                .or. parton_is_branched (prt%child2)) .and. &
                (space_part_norm (prt%momentum) * space_part_norm &
                (prt%child1%momentum) > zero)) then
              costheta = (space_part (prt%momentum) * &
                   space_part(prt%child1%momentum)) / &
                   (space_part_norm (prt%momentum) * &
                    space_part_norm (prt%child1%momentum))
           else
              costheta = -two
           end if
        else
           costheta = -two
        end if
      end function parton_get_costheta_motherfirst
```

Get the parton velocities.

⟨*Shower partons: public*⟩+≡
```
      public :: parton_get_beta
```

⟨*Shower partons: procedures*⟩+≡
```
      function get_beta (t,E) result (beta)
        real(default), intent(in) :: t,E
        real(default) :: beta
        beta = sqrt (max (1.E-6_default, one - t /(E**2)))
```

```
      end function get_beta

      function parton_get_beta (prt) result (beta)
        type(parton_t), intent(in) :: prt
        real(default) :: beta
        beta = get_beta (prt%t, vector4_get_component (prt%momentum, 0))
      end function parton_get_beta
```

Write routine.

⟨*Shower partons: public*⟩+≡
```
      public :: parton_write
```

⟨*Shower partons: procedures*⟩+≡
```
      subroutine parton_write (prt, unit)
        type(parton_t), intent(in) :: prt
        integer, intent(in), optional :: unit
        integer :: u
        u = output_unit (unit); if (u < 0) return

        write (u, "(1x,4A)", advance ="no")  "Shower parton <nr>", &
             TAB, "<type>", TAB
        write (u, "(1x,3A)") "<parent>", TAB, "<mom(0:3)>"
        write (u, "(2x,I5,3A)", advance = "no")  prt%nr, TAB, TAB, TAB
        if (parton_is_final (prt)) then
           write (u, "(1x,I5,1x,A)", advance = "no") prt%type, TAB
        else
           write (u, "('[',I5,']',A)", advance = "no") prt%type, TAB
        end if
        if (associated (prt%parent)) then
           write (u, "(I5,2A)", advance = "no") prt%parent%nr, TAB, TAB
        else
           write (u, "(5x,2A)", advance = "no") TAB, TAB
        end if
        write (u, "(4(ES12.5,A))") &
             vector4_get_component (prt%momentum, 0), TAB, &
             vector4_get_component (prt%momentum, 1), TAB, &
             vector4_get_component (prt%momentum, 2), TAB, &
             vector4_get_component (prt%momentum, 3)
        write (u, "(1x,5A)", advance = "no") "<p4square>", TAB, TAB, "<t>", TAB
        write (u, "(1x,7A)") TAB, "<scale>", TAB, TAB, "<c1>", TAB, "<c2>"
        write (u, "(3(ES12.5,A))", advance = "no") &
             parton_p4square(prt), TAB, prt%t, TAB // TAB, prt%scale, TAB
        write (u, "(2(I4,A))") prt%c1, TAB, prt%c2, TAB
        if (parton_is_branched (prt)) then
           if (prt%belongstoFSR) then
              write(u, "(3x,5(ES8.5,1x),A1)", advance = "no") &
                   parton_get_costheta (prt), &
                   parton_get_costheta_correct (prt), prt%costheta, prt%z, &
                   parton_get_costheta_motherfirst (prt), 'b'
           else
              write(u, "(3x,5(ES8.5,1x),A1)", advance = "no") prt%z, prt%x, &
                   parton_get_costheta_correct (prt), prt%costheta, &
                   parton_get_costheta_motherfirst (prt), 'b'
           end if
```

```
            else
               if (prt%belongstoFSR) then
                  write (u, "(43x)", advance = "no")
               else
                  write (u, "(9x,ES8.5,26x)", advance = "no")  prt%x
               end if
            end if
            write (u, "(A)", advance = "no") " Parton "
            if (prt%belongstoFSR) then
               write (u, "(A)", advance = "no")  "is FSR, "
            else
               if (associated (prt%initial)) then
                  write (u, "(A,I1)", advance = "no")  "from hadron, ", prt%initial%nr
               else
                  write (u, "(A)", advance = "no")   " "
               end if
            end if
            if (parton_is_final (prt)) then
               write (u, "(A)", advance = "no")  "is final, "
            else
               write (u, "(A)", advance = "no")   " "
            end if
            if (parton_is_simulated (prt)) then
               write (u, "(A)", advance = "no")  "is simulated, "
            else
               write (u, "(A)", advance = "no")   " "
            end if
            if (associated (prt%child1) .and. associated (prt%child2)) then
               write (u, "(A,2(I5),A)", advance = "no") &
                     "has children: ", prt%child1%nr, prt%child2%nr, ", "
            else if (associated (prt%child1)) then
               write (u, "(A,1(I5),A)", advance = "no") &
                     "has children: ", prt%child1%nr, ", "
            end if
            if (prt%belongstointeraction) then
               write (u, "(A,I2)") "belongs to        ", &
                     prt%interactionnr
            else
               write (u, "(A,I2)") "does not belong to ", &
                     prt%interactionnr
            end if
            write (u,"(A)")  TAB
         end subroutine parton_write
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_final
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_final (prt) result (is_final)
      type(parton_t), intent(in) :: prt
      logical :: is_final
      is_final = .false.
      if (prt%belongstoFSR) then
         is_final = .not. associated (prt%child1) .and. &
```

2157

```
                (.not. prt%belongstointeraction .or. &
                (prt%belongstointeraction .and. prt%simulated))
           end if
        end function parton_is_final
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_branched
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_branched (prt) result (is_branched)
      type(parton_t), intent(in) :: prt
      logical :: is_branched
      is_branched = associated (prt%child1) .and. associated (prt%child2)
    end function parton_is_branched
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_set_simulated
```

⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_set_simulated (prt, sim)
      type(parton_t), intent(inout) :: prt
      logical, intent(in), optional :: sim
      if (present (sim)) then
         prt%simulated = sim
      else
         prt%simulated = .true.
      end if
    end subroutine parton_set_simulated
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_simulated
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_simulated (prt) result (is_simulated)
      type(parton_t), intent(in) :: prt
      logical :: is_simulated
      is_simulated = prt%simulated
    end function parton_is_simulated
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_get_momentum
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_get_momentum (prt, i) result (mom)
      type(parton_t), intent(in) :: prt
      integer, intent(in) :: i
      real(default) :: mom
      select case (i)
      case (0)
         mom = vector4_get_component (prt%momentum,0)
      case (1)
         mom = vector4_get_component (prt%momentum,1)
      case (2)
         mom = vector4_get_component (prt%momentum,2)
```

```
      case (3)
         mom = vector4_get_component (prt%momentum,3)
      case default
         mom = 0
      end select
    end function parton_get_momentum
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_set_momentum
```

⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_set_momentum (prt, EE, ppx, ppy, ppz)
      type(parton_t), intent(inout) :: prt
      real(default), intent(in) :: EE, ppx, ppy, ppz
      prt%momentum = vector4_moving &
           (EE, vector3_moving ([ppx, ppy, ppz]))
    end subroutine parton_set_momentum
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_set_energy
```

⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_set_energy (prt, E)
      type(parton_t), intent(inout) :: prt
      real(default), intent(in) :: E
      call vector4_set_component (prt%momentum, 0, E)
    end subroutine parton_set_energy
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_get_energy
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_get_energy (prt) result (E)
      type(parton_t), intent(in) :: prt
      real(default) :: E
      E = vector4_get_component (prt%momentum, 0)
    end function parton_get_energy
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_set_parent
```

⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_set_parent (prt, parent)
      type(parton_t), intent(inout) :: prt
      type(parton_t), intent(in) , target :: parent
      prt%parent => parent
    end subroutine parton_set_parent
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_get_parent
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_get_parent (prt) result (parent)
      type(parton_t), intent(in) :: prt
      type(parton_t), pointer :: parent
      parent => prt%parent
    end function parton_get_parent
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_set_initial
```
⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_set_initial (prt, initial)
      type(parton_t), intent(inout) :: prt
      type(parton_t), intent(in) , target :: initial
      prt%initial => initial
    end subroutine parton_set_initial
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_get_initial
```
⟨*Shower partons: procedures*⟩+≡
```
    function parton_get_initial (prt) result (initial)
      type(parton_t), intent(in) :: prt
      type(parton_t), pointer :: initial
      initial => prt%initial
    end function parton_get_initial
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_set_child
```
⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_set_child (prt, child, i)
      type(parton_t), intent(inout) :: prt
      type(parton_t), intent(in), target :: child
      integer, intent(in) ::  i
      if (i == 1) then
         prt%child1 => child
      else
         prt%child2 => child
      end if
    end subroutine parton_set_child
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_get_child
```
⟨*Shower partons: procedures*⟩+≡
```
    function parton_get_child (prt, i) result (child)
      type(parton_t), intent(in) :: prt
      integer, intent(in) :: i
      type(parton_t), pointer :: child
      child => null ()
      if (i == 1) then
         child => prt%child1
      else
```

2160

```
      child => prt%child2
    end if
  end function parton_get_child
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_quark
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_quark (prt) result (is_quark)
      type(parton_t), intent(in) ::prt
      logical :: is_quark
      is_quark= abs (prt%type) <= 6 .and. prt%type /= 0
    end function parton_is_quark
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_squark
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_squark (prt) result (is_squark)
      type(parton_t), intent(in) ::prt
      logical :: is_squark
      is_squark = ((abs(prt%type) >= 1000001) .and. (abs(prt%type) <= 1000006)) &
             .or. ((abs(prt%type) >= 2000001) .and. (abs(prt%type) <= 2000006))
    end function parton_is_squark
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_gluon
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_gluon (prt) result (is_gluon)
      type(parton_t), intent(in) :: prt
      logical :: is_gluon
      is_gluon = prt%type == 21 .or. prt%type == 9
    end function parton_is_gluon
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_gluino
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_gluino (prt) result (is_gluino)
      type(parton_t), intent(in) :: prt
      logical :: is_gluino
      is_gluino = prt%type == 1000021
    end function parton_is_gluino
```

Only the proton is implemented yet.

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_hadron
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_hadron (prt) result (is_hadron)
      type(parton_t), intent(in) :: prt
      logical :: is_hadron
      is_hadron = abs (prt%type) == 2212
```

```
    end function parton_is_hadron
```

TODO: SUSY partons.

⟨*Shower partons: public*⟩+≡
```
    public :: parton_is_colored
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_is_colored (prt) result (is_colored)
      type(parton_t), intent(in) ::prt
      logical :: is_colored
      is_colored = parton_is_quark (prt) .or. parton_is_gluon (prt)
    end function parton_is_colored
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_p4square
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_p4square (prt) result (p4square)
      type(parton_t), intent(in) :: prt
      real(default) :: p4square
      p4square = prt%momentum**2
    end function parton_p4square
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_p3square
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_p3square (prt) result (p3square)
      type(parton_t), intent(in) :: prt
      real(default) :: p3square
      p3square = parton_p3abs (prt)**2
    end function parton_p3square
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_p3abs
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_p3abs (prt) result (p3abs)
      type(parton_t), intent(in) :: prt
      real(default) :: p3abs
      p3abs = space_part_norm (prt%momentum)
    end function parton_p3abs
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_mass
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_mass (prt) result (mass)
      type(parton_t), intent(in) :: prt
      real(default) :: mass
      mass = mass_type (prt%type)
    end function parton_mass
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_mass_squared
```

⟨*Shower partons: procedures*⟩+≡
```
    function parton_mass_squared (prt) result (mass_squared)
      type(parton_t), intent(in) :: prt
      real(default) :: mass_squared
      mass_squared = mass_squared_type (prt%type)
    end function parton_mass_squared
```

⟨*Shower partons: public*⟩+≡
```
    public :: P_prt_to_child1
```

⟨*Shower partons: procedures*⟩+≡
```
    function P_prt_to_child1 (prt) result (retvalue)
      type(parton_t), intent(in) :: prt
      real(default) :: retvalue
      retvalue = zero
      if (parton_is_gluon (prt)) then
         if (parton_is_quark (prt%child1)) then
            retvalue = P_gqq (prt%z)
         else if (parton_is_gluon (prt%child1)) then
            retvalue = P_ggg (prt%z) + P_ggg (one - prt%z)
         end if
      else if (parton_is_quark (prt)) then
         if (parton_is_quark (prt%child1)) then
            retvalue = P_qqg (prt%z)
         else if (parton_is_gluon (prt%child1)) then
            retvalue = P_qqg (one - prt%z)
         end if
      end if
    end function P_prt_to_child1
```

This function returns whether the kinematics of the branching of parton `prt` into its daughters are allowed or not.

⟨*Shower partons: public*⟩+≡
```
    public :: thetabar
```

⟨*Shower partons: procedures*⟩+≡
```
    function thetabar (prt, recoiler, E3out) result (retvalue)
      type(parton_t), intent(inout) :: prt
      type(parton_t), intent(in) :: recoiler
      real(default), intent(out), optional :: E3out
      logical :: retvalue
      real(default) :: ctheta, cthetachild1
      real(default) p1, p4, p3, E3, shat

      shat = (prt%child1%momentum + recoiler%momentum)**2
      E3 = 0.5_default * (shat / prt%z -recoiler%t + prt%child1%t - &
           parton_mass_squared (prt%child2)) / sqrt(shat)
      if (present (E3out)) then
         E3out = E3
      end if
      !!! absolute values of momenta in a 3 -> 1 + 4 branching
```

2163

```
        p3 = sqrt (E3**2 - prt%t)
        p1 = sqrt (parton_get_energy (prt%child1)**2 - prt%child1%t)
        p4 = sqrt (max (zero, (E3 - parton_get_energy (prt%child1))**2 &
             - prt%child2%t))
        if (p3 > zero) then
            retvalue = ((p1 + p4 >= p3) .and. (p3 >= abs(p1 - p4)) )
            if (retvalue .and. isr_angular_ordered) then
                !!! check angular ordering
                if (associated (prt%child1)) then
                    if (associated (prt%child1%child2)) then
                        ctheta = (E3**2 - p1**2 - p4**2 +prt%t) / (two * p1 * p4)
                        cthetachild1 = (parton_get_energy (prt%child1)**2 - &
                            space_part (prt%child1%child1%momentum)**2 &
                            - space_part (prt%child1%child2%momentum)**2 + prt%child1%t) &
                            / (two * space_part (prt%child1%child1%momentum)**1 * &
                            space_part(prt%child1%child2%momentum)**1)
                        retvalue= (ctheta > cthetachild1)
                    end if
                end if
            end if
        else
            retvalue = .false.
        end if
    end function thetabar
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_apply_z
```

⟨*Shower partons: procedures*⟩+≡
```
    recursive subroutine parton_apply_z(prt, newz)
      type(parton_t), intent(inout) :: prt
      real(default), intent(in) :: newz
      if (D_print) print *, "old z:", prt%z , " new z: ", newz
      prt%z = newz
      if (associated (prt%child1) .and. associated (prt%child2)) then
          call parton_set_energy (prt%child1, newz * parton_get_energy (prt))
          call parton_apply_z (prt%child1, prt%child1%z)
          call parton_set_energy (prt%child2, (1.-newz) * parton_get_energy (prt))
          call parton_apply_z (prt%child2, prt%child2%z)
      end if
    end subroutine parton_apply_z
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_apply_costheta
```

⟨*Shower partons: procedures*⟩+≡
```
    recursive subroutine parton_apply_costheta (prt)
      type(parton_t), intent(inout) :: prt
      prt%z = 0.5_default * (one + parton_get_beta (prt) * prt%costheta)
      if (associated (prt%child1) .and. associated (prt%child2) ) then
          if (parton_is_simulated (prt%child1) .and. &
              parton_is_simulated (prt%child2)) then
              prt%z = 0.5_default * (one + (prt%child1%t - prt%child2%t) / &
                  prt%t + parton_get_beta (prt) * prt%costheta * &
```

```
                    sqrt((prt%t - prt%child1%t - prt%child2%t)**2 - &
                    4 * prt%child1%t * prt%child2%t) / prt%t)
              if (prt%type /= 94) then
                 call parton_set_energy (prt%child1, &
                       prt%z * parton_get_energy (prt))
                 call parton_set_energy (prt%child2, (one - prt%z) * &
                       parton_get_energy (prt))
              end if
              call parton_generate_ps (prt)
              call parton_apply_costheta (prt%child1)
              call parton_apply_costheta (prt%child2)
           end if
        end if
    end subroutine parton_apply_costheta
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_apply_lorentztrafo
```

⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_apply_lorentztrafo (prt, L)
      type(parton_t), intent(inout) :: prt
      type(lorentz_transformation_t), intent(in) :: L
      prt%momentum = L * prt%momentum
    end subroutine parton_apply_lorentztrafo
```

⟨*Shower partons: public*⟩+≡
```
    public :: parton_apply_lorentztrafo_recursive
```

⟨*Shower partons: procedures*⟩+≡
```
    recursive subroutine parton_apply_lorentztrafo_recursive (prt, L)
      type(parton_t), intent(inout) :: prt
      type(lorentz_transformation_t) ,intent(in) :: L
      if (prt%type /= 2212 .and. prt%type /= 9999) then
         !!! don't boost hadrons and beam-remnants
         call parton_apply_lorentztrafo (prt, L)
      end if
      if (associated (prt%child1) .and. associated (prt%child2)) then
         if ((parton_p3abs(prt%child1) == zero) .and. &
             (parton_p3abs(prt%child2) == zero) .and. &
             (.not. prt%child1%belongstointeraction) .and. &
             (.not. prt%child2%belongstointeraction)) then
            !!! don't boost unevolved timelike partons
         else
            call parton_apply_lorentztrafo_recursive (prt%child1, L)
            call parton_apply_lorentztrafo_recursive (prt%child2, L)
         end if
      else
         if (associated (prt%child1)) then
            call parton_apply_lorentztrafo_recursive (prt%child1, L)
         end if
         if (associated (prt%child2)) then
            call parton_apply_lorentztrafo_recursive (prt%child2, L)
         end if
      end if
```

```
      end subroutine parton_apply_lorentztrafo_recursive
```

This takes the three-momentum of a parton and generates three-momenta of its children.

⟨Shower partons: public⟩+≡
```
    public :: parton_generate_ps
```

⟨Shower partons: procedures⟩+≡
```
    subroutine parton_generate_ps (prt)
      type(parton_t), intent(inout) :: prt
      real(default), dimension(1:3, 1:3) :: directions
      integer i,j
      real(default) :: scproduct, pabs, p1abs, p2abs, x, ptabs, phi
      real(default), dimension(1:3) :: momentum

      type(vector3_t) :: pchild1_direction
      type(lorentz_transformation_t) :: L, rotation

      if (D_print) print *, " generate_ps for parton " , prt%nr
      if (.not. (associated (prt%child1) .and. associated (prt%child2))) then
         print *, "no children for generate_ps"
         return
      end if
      !!! test if parton is a virtual parton from the imagined parton shower history
      if (prt%type == 94) then
         L = inverse (boost (prt%momentum, sqrt(prt%t)))
         !!! boost to restframe of mother
         call parton_apply_lorentztrafo (prt, L)
         call parton_apply_lorentztrafo (prt%child1, L)
         call parton_apply_lorentztrafo (prt%child2, L)
         !!! Store child1's momenta
         pchild1_direction = direction (space_part (prt%child1%momentum))
         !!! Redistribute energy
         call parton_set_energy (prt%child1, (parton_get_energy (prt)**2- &
              prt%child2%t + prt%child1%t) / (two * parton_get_energy (prt)))
         call parton_set_energy (prt%child2, parton_get_energy (prt) - &
              parton_get_energy (prt%child1))

         ! rescale momenta and set momenta to be along z-axis
         prt%child1%momentum = vector4_moving (parton_get_energy (prt%child1), &
              vector3_canonical(3) * sqrt(parton_get_energy (prt%child1)**2 - &
              prt%child1%t))
         prt%child2%momentum = vector4_moving (parton_get_energy (prt%child2), &
              vector3_canonical(3) * (-sqrt(parton_get_energy (prt%child2)**2 - &
              prt%child2%t)))

         !!! rotate so that total momentum is along former total momentum
         rotation = rotation_to_2nd (space_part (prt%child1%momentum), &
              pchild1_direction)
         call parton_apply_lorentztrafo (prt%child1, rotation)
         call parton_apply_lorentztrafo (prt%child2, rotation)

         L = inverse (L)               !!! inverse of the boost to restframe of mother
         call parton_apply_lorentztrafo (prt, L)
```

```
      call parton_apply_lorentztrafo (prt%child1, L)
      call parton_apply_lorentztrafo (prt%child2, L)
else
   !!! directions(1,:) -> direction of the parent parton
   if (parton_p3abs (prt) == zero) return
   do i = 1, 3
      directions(1,i) = parton_get_momentum (prt,i) / parton_p3abs (prt)
   end do
   !!! directions(2,:) and directions(3,:) -> two random directions
   !!!    perpendicular to the direction of the parent parton
   do i = 1, 3
      do j = 2, 3
         call tao_random_number (directions(j,i))
      end do
   end do
   do i = 2, 3
      scproduct = zero
      do j = 1, i - 1
         scproduct = directions(i,1) * directions(j,1) + &
            directions(i,2) * directions(j,2) + directions(i,3) * directions(j,3)
         directions(i,1) = directions(i,1) - directions(j,1) * scproduct
         directions(i,2) = directions(i,2) - directions(j,2) * scproduct
         directions(i,3) = directions(i,3) - directions(j,3) * scproduct
      end do
      scproduct = directions(i,1)**2 + directions(i,2)**2 + directions(i,3)**2
      do j = 1, 3
         directions(i,j) = directions(i,j) / sqrt(scproduct)
      end do
   end do
   ⟨Enforce right-handed system for directions⟩

   pabs = parton_p3abs(prt)
   if ((parton_get_energy (prt%child1)**2 - prt%child1%t < 0) .or. &
       (parton_get_energy (prt%child2)**2 - prt%child2%t < 0)) then
      if (D_print) print *, "err: error at generate_ps(), E^2 < t"
      return
   end if
   p1abs = sqrt (parton_get_energy (prt%child1)**2 - prt%child1%t)
   p2abs = sqrt (parton_get_energy (prt%child2)**2 - prt%child2%t)
   x = (pabs**2 + p1abs**2 - p2abs**2) / (two * pabs)
   if (pabs > p1abs + p2abs .or. &
       pabs < abs(p1abs - p2abs)) then
      if (D_print) then
         print *,"error at generate_ps, Dreiecksungleichung for parton ", &
                 prt%nr, " ", parton_p3abs(prt)," ",p1abs," ",p2abs
         call parton_write (prt)
         call parton_write (prt%child1)
         call parton_write (prt%child2)
      end if
      return
   end if
   !!! Due to numerical problems transverse momentum could be imaginary ->
   !!!    set transverse momentum to zero
   ptabs = sqrt (max (p1abs * p1abs - x * x, zero))
```

2167

```
        call tao_random_number (phi)
        phi = twopi * phi
        do i = 1, 3
            momentum(i) = x * directions(1,i) + ptabs * &
                    (cos(phi) * directions(2,i) + sin(phi) * directions(3,i))
        end do
        call parton_set_momentum (prt%child1, parton_get_energy (prt%child1), &
            momentum(1), momentum(2), momentum(3))
        do i = 1, 3
            momentum(i) = (parton_p3abs(prt) - x) * directions(1,i) - &
                    ptabs * (cos(phi) * directions(2,i) + sin(phi) * directions(3,i))
        end do
        call parton_set_momentum (prt%child2, parton_get_energy (prt%child2), &
            momentum(1), momentum(2), momentum(3))
     end if
   end subroutine parton_generate_ps
```

⟨*Enforce right-handed system for* `directions`⟩≡
```
  if ((directions(1,1) * (directions(2,2) * directions(3,3) - &
       directions(2,3) * directions(3,2)) + &
       directions(1,2) * (directions(2,3) * directions(3,1) - &
       directions(2,1) * directions(3,3)) + &
       directions(1,3) * (directions(2,1) * directions(3,2) - &
       directions(2,2) * directions(3,1))) < 0) then
     directions(3,1) = - directions(3,1)
     directions(3,2) = - directions(3,2)
     directions(3,3) = - directions(3,3)
  end if
```

This routine is similar to `parton_generate_ps`, but now for the ISR. It takes the three-momentum of a parton's first child as fixed and generates the two remaining three-momenta.

⟨*Shower partons: public*⟩+≡
```
    public :: parton_generate_ps_ini
```

⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_generate_ps_ini(prt)
      type(parton_t), intent(inout) :: prt
      real(default), dimension(1:3, 1:3) :: directions
      integer :: i,j
      real(default) :: scproduct, pabs, p1abs, p2abs, x, ptabs, phi
      real(default), dimension(1:3) :: momentum

      if (D_print) print *, " generate_ps_ini for parton " , prt%nr
      if (.not. (associated(prt%child1) .and. associated(prt%child2))) then
         print *, "error in parton_generate_ps_ini"
         return
      end if

      if (parton_is_hadron(prt) .eqv. .false.) then
         !!! generate ps for normal partons
         do i = 1, 3
            directions(1,i) = parton_get_momentum (prt%child1,i) / &
                    parton_p3abs(prt%child1)
```

2168

```
      end do
      do i = 1, 3
         do j = 2, 3
            call tao_random_number (directions(j,i))
         end do
      end do
      do i = 2, 3
         scproduct = zero
         do j = 1, i - 1
            scproduct = directions(i,1) * directions(j,1) + &
                 directions(i,2) * directions(j,2) + directions(i,3) * directions(j,3)
            directions(i,1) = directions(i,1) - directions(j,1) * scproduct
            directions(i,2) = directions(i,2) - directions(j,2) * scproduct
            directions(i,3) = directions(i,3) - directions(j,3) * scproduct
         end do
         scproduct = directions(i,1)**2 + directions(i,2)**2 + directions(i,3)**2
         do j = 1, 3
            directions(i,j) = directions(i,j) / sqrt(scproduct)
         end do
      end do
      ⟨Enforce right-handed system for directions⟩

      pabs = parton_p3abs (prt%child1)
      p1abs = sqrt (parton_get_energy (prt)**2 - prt%t)
      p2abs = sqrt (max(zero, parton_get_energy (prt%child2)**2 - &
           prt%child2%t))

      x = (pabs**2 + p1abs**2 - p2abs**2) / (two * pabs)
      if (pabs > p1abs + p2abs .or. pabs < abs(p1abs - p2abs)) then
         print *, "error at generate_ps, Dreiecksungleichung for parton ", &
              prt%nr, " ", pabs," ",p1abs," ",p2abs
         call parton_write (prt)
         call parton_write (prt%child1)
         call parton_write (prt%child2)
         return
      end if
      if (D_print) print *, "x:",x
      ptabs = sqrt (p1abs * p1abs - x**2)
      call tao_random_number (phi)
      phi = twopi * phi
      do i = 1,3
         momentum(i) = x * directions(1,i) + ptabs * (cos(phi) * &
              directions(2,i) + sin(phi) * directions(3,i))
      end do
      call parton_set_momentum (prt, parton_get_energy(prt), &
           momentum(1), momentum(2), momentum(3))
      do i = 1, 3
         momentum(i) = (x - pabs) * directions(1,i) + ptabs * (cos(phi) * &
              directions(2,i) + sin(phi) * directions(3,i))
      end do
      call parton_set_momentum (prt%child2, parton_get_energy(prt%child2), &
           momentum(1), momentum(2), momentum(3))
   else
      !!! for first partons just set beam remnants momentum
```

2169

```
            prt%child2%momentum = prt%momentum - prt%child1%momentum
        end if
    end subroutine parton_generate_ps_ini
```

16.2.3   The analytic FSR

⟨*Shower partons: public*⟩+≡
```
    public :: parton_next_t_ana
```
⟨*Shower partons: procedures*⟩+≡
```
    subroutine parton_next_t_ana (prt)
      type(parton_t), intent(inout) :: prt
      integer :: gtoqq
      real(default) :: integral, random

      if (signal_is_pending ()) return
      if (D_print) then
         print *, "next_t_ana for parton " , prt%nr
      end if

      ! check if branchings are possible at all
      if (min (prt%t, parton_get_energy(prt)**2) < &
           parton_mass_squared(prt) + D_Min_t) then
        prt%t = parton_mass_squared (prt)
        call parton_set_simulated (prt)
        return
      end if

      integral = zero
      call tao_random_number (random)
      do
         if (signal_is_pending ()) return
         call parton_simulate_stept (prt, integral, random, gtoqq, .false.)
         if (parton_is_simulated (prt)) then
            if (parton_is_gluon (prt)) then
               !!! Abusing the x-variable to store the information to which
               !!! quark flavour the gluon branches (if any)
               prt%x = one * gtoqq + 0.1_default
               !!! x = gtoqq + 0.1 -> int(x) will be the quark flavour or
               !!! zero for g -> gg
            end if
            exit
         end if
      end do
    end subroutine parton_next_t_ana
```

⟨*Shower partons: procedures*⟩+≡
```
    function cmax (prt, tt) result (cma)
      type(parton_t), intent(in) :: prt
      real(default), intent(in), optional :: tt
      real(default) :: cma
      real(default) :: t, cost
```

```
      if (present(tt)) then
         t = tt
      else
         t = prt%t
      end if

      if (associated (prt%parent)) then
         cost = parton_get_costheta (prt%parent)
         cma = min (0.99999_default, sqrt( max(zero, one - t/ &
               (parton_get_beta(prt) *parton_get_energy(prt))**2 * &
               (one + cost) / (one - cost))))
      else
         cma = 0.99999_default
      end if
   end function cmax
```

Simulation routine. The variable `lookatsister` takes constraints from the sister parton into account, if not given it is assumed `.true.`. `a` and `x` are three-dimensional arrays for values used for the integration.

⟨*Shower partons: public*⟩+≡
```
   public :: parton_simulate_stept
```

⟨*Shower partons: procedures*⟩+≡
```
   subroutine parton_simulate_stept &
       (prt, integral, random, gtoqq, lookatsister)
     type(parton_t), intent(inout) :: prt
     real(default), intent(inout) :: integral
     real(default), intent(inout) :: random
     integer, intent(out) :: gtoqq
     logical, intent(in), optional :: lookatsister

     type(parton_t), pointer :: sister
     real(default) :: tstep, tmin, oldt
     real(default) :: c, cstep
     real(default), dimension(3) :: z, P
     real(default) :: to_integral
     real(default) :: a11,a12,a13,a21,a22,a23
     real(default) :: cmax_t
     real(default) :: temprand
     real(default), dimension(3) :: a, x

     ! higher values -> faster but coarser
     real(default), parameter :: tstepfactor = 0.02_default
     real(default), parameter :: tstepmin = 0.5_default
     real(default), parameter :: cstepfactor = 0.8_default
     real(default), parameter :: cstepmin = 0.03_default

     if (signal_is_pending ()) return
     gtoqq = 111 ! illegal value
     call parton_set_simulated (prt, .false.)

     ⟨Set sister if lookatsister is true or not given⟩

     tmin = D_Min_t + parton_mass_squared (prt)
```

```fortran
if (parton_is_quark(prt)) then
   to_integral = three *pi * log(one / random)
else if (parton_is_gluon(prt)) then
   to_integral = four *pi * log(one / random)
else
   prt%t = parton_mass_squared (prt)
   call parton_set_simulated (prt)
   return
end if

if (associated (sister)) then
   if (sqrt(prt%t) > sqrt(prt%parent%t) - &
         sqrt(parton_mass_squared (sister))) then
      prt%t = (sqrt (prt%parent%t) - sqrt (parton_mass_squared (sister)))**2
   end if
end if
if (prt%t > parton_get_energy(prt)**2) then
    prt%t = parton_get_energy(prt)**2
end if

if (prt%t <= tmin) then
   prt%t = parton_mass_squared (prt)
   call parton_set_simulated (prt)
   return
end if

! simulate the branchings between prt%t and prt%t - tstep
tstep = max(tstepfactor * (prt%t - 0.9_default * tmin), tstepmin)
cmax_t = cmax(prt)
c = - cmax_t ! take highest t -> minimal constraint
cstep = max(cstepfactor * (one - abs(c)), cstepmin)
! get values at border of "previous" bin -> to be used in first bin
z(3) = 0.5_default + 0.5_default * get_beta (prt%t - &
     0.5_default * tstep, parton_get_energy (prt)) * c
if (parton_is_gluon (prt)) then
   P(3) = P_ggg (z(3)) + P_gqq (z(3)) * number_of_flavors (prt%t)
else
   P(3) = P_qqg (z(3))
end if
a(3) = D_alpha_s_fsr (z(3) * (one - z(3)) * prt%t) * P(3) / &
     (prt%t - 0.5_default * tstep)

do while (c < cmax_t .and. (integral < to_integral))
   if (signal_is_pending ()) return
   cmax_t = cmax (prt)
   cstep = max (cstepfactor * (one - abs(c)**2), cstepmin)
   if (c + cstep > cmax_t) then
      cstep = cmax_t - c
   end if
   if (cstep < 1E-9_default) then
      !!! reject too small bins
      exit
   end if
   z(1) = z(3)
```

```
z(2) = 0.5_default + 0.5_default * get_beta &
    (prt%t - 0.5_default * tstep, parton_get_energy (prt)) * &
    (c + 0.5_default * cstep)
z(3) = 0.5_default + 0.5_default * get_beta &
    (prt%t - 0.5_default * tstep, parton_get_energy (prt)) * (c + cstep)
P(1) = P(3)
if (parton_is_gluon (prt)) then
    P(2) = P_ggg(z(2)) + P_gqq(z(2)) * number_of_flavors (prt%t)
    P(3) = P_ggg(z(3)) + P_gqq(z(3)) * number_of_flavors (prt%t)
else
    P(2) = P_qqg(z(2))
    P(3) = P_qqg(z(3))
end if
! get values at borders of the intgral and in the middle
a(1) = a(3)
a(2) = D_alpha_s_fsr(z(2) * (one - z(2)) * prt%t) * P(2) / &
    (prt%t - 0.5_default * tstep)
a(3) = D_alpha_s_fsr(z(3) * (one - z(3)) * prt%t) * P(3) / &
    (prt%t - 0.5_default * tstep)

! fit x(1) + x(2)/(1 + c) + x(3)/(1 - c) to these values !! a little tricky
a11 = (one+c+0.5_default*cstep) * (one-c-0.5_default*cstep) - &
    (one-c) * (one+c+0.5_default*cstep)
a12 = (one-c-0.5_default*cstep) - (one+c+0.5_default*cstep) * &
    (one-c) / (one+c)
a13 = a(2) * (one+c+0.5_default*cstep) * (one-c-0.5_default*cstep) - &
    a(1) * (one-c) * (one+c+0.5_default*cstep)
a21 = (one+c+cstep) * (one-c-cstep) - (one+c+cstep) * (one-c)
a22 = (one-c-cstep) - (one+c+cstep) * (one-c) / (one+c)
a23 = a(3) * (one+c+cstep) * (one-c-cstep) - &
    a(1) * (one-c) * (one+c+cstep)

x(2) = (a23 - a21 * a13 / a11) / (a22 - a12 * a21 / a11)
x(1) = (a13 - a12 * x(2)) / a11
x(3) = a(1) * (one - c) - x(1) * (one - c) - x(2) * (one - c) / (one + c)

integral = integral + tstep * (x(1) * cstep + x(2) * &
    log((one + c + cstep) / (one + c)) - x(3) * &
    log((one - c - cstep) / (one - c)))

if (integral > to_integral) then
    oldt = prt%t
    call tao_random_number (temprand)
    prt%t = prt%t - temprand * tstep
    call tao_random_number (temprand)
    prt%costheta = c + (0.5_default - temprand) * cstep
    call parton_set_simulated (prt)

    if (prt%t < D_Min_t + parton_mass_squared(prt)) then
        prt%t = parton_mass_squared (prt)
    end if
    if (prt%costheta.lt.-cmax_t .or. prt%costheta.gt.cmax_t) then
        ! reject branching due to violation of costheta-limits
        call tao_random_number (random)
```

```
               if (parton_is_quark (prt)) then
                  to_integral = three * pi * log(one / random)
               else if (parton_is_gluon(prt)) then
                  to_integral = four * pi * log(one / random)
               end if
               integral = zero
               prt%t = oldt
               call parton_set_simulated (prt, .false.)
            end if
            if (parton_is_gluon (prt)) then
               ! decide between g->gg and g->qqbar splitting
               z(1) = 0.5_default + 0.5_default * prt%costheta
               call tao_random_number (temprand)
               if (P_ggg(z(1)) > temprand * (P_ggg (z(1)) + P_gqq (z(1)) * &
                    number_of_flavors(prt%t))) then
                  gtoqq = 0
               else
                  call tao_random_number (temprand)
                  gtoqq = 1 + temprand * number_of_flavors (prt%t)
               end if
            end if
         else
            c = c + cstep
         end if
         cmax_t = cmax (prt)
      end do
      if (integral <= to_integral) then
         prt%t = prt%t - tstep
         if (prt%t < D_Min_t + parton_mass_squared (prt)) then
            prt%t = parton_mass_squared (prt)
            call parton_set_simulated(prt)
         end if
      end if
    end if
  end subroutine parton_simulate_stept
```

⟨*Set* `sister` *if* `lookatsister` *is true or not given*⟩≡
```
  sister => null()
  SET_SISTER: do
     if (present (lookatsister)) then
        if (.not. lookatsister) then
           exit SET_SISTER
        end if
     end if
     if (prt%nr == prt%parent%child1%nr) then
        sister => prt%parent%child2
     else
        sister => prt%parent%child1
     end if
     exit SET_SISTER
  end do SET_SISTER
```

From the whole ISR algorithm all functionality has been moved to `shower_core.f90`. Only `maxzz` remains here, because more than one module needs to access it.

⟨*Shower partons: public*⟩+≡

```
      public :: maxzz
```
⟨*Shower partons: procedures*⟩+≡
```
    function maxzz (shat, s) result (maxz)
      real(default), intent(in) :: shat,s
      real(default) :: maxz
      maxz= min (maxz_isr, one - (two * minenergy_timelike * sqrt(shat)) / s)
    end function maxzz
```

16.3   Main shower module

⟨`shower_core.f90`⟩≡
  ⟨*File header*⟩

```
  module shower_core
```

  ⟨*Use kinds*⟩
```
    use constants !NODEP!
    use limits !NODEP!
    use lorentz !NODEP!
    use diagnostics !NODEP!
    use file_utils !NODEP!
    use tao_random_numbers !NODEP!
    use shower_base
    use shower_partons
    use ckkw_pseudo_weights
```

  ⟨*Standard module head*⟩

  ⟨*Shower core: public*⟩

  ⟨*Shower core: parameters*⟩

  ⟨*Shower core: types*⟩

```
  contains
```

  ⟨*Shower core: procedures*⟩

```
  end module shower_core
```

⟨*Shower core: public*⟩≡
```
    public :: shower_interaction_t
```
⟨*Shower core: types*⟩≡
```
    type :: shower_interaction_t
       type(parton_pointer_t), dimension(:), allocatable :: partons
    end type shower_interaction_t

    type :: shower_interaction_pointer_t
       type(shower_interaction_t), pointer :: i => null ()
    end type shower_interaction_pointer_t
```

The main type of `shower_core`:

⟨*Shower core: public*⟩+≡
```
    public :: shower_t
```

⟨*Shower core: types*⟩+≡
```
    type :: shower_t
       type(shower_interaction_pointer_t), dimension(:), allocatable :: &
            interactions
       type(parton_pointer_t), dimension(:), allocatable :: partons
       integer :: next_free_nr
       integer :: next_color_nr
       logical :: valid
     contains
     ⟨Shower core: shower: TBP⟩
    end type shower_t
```

The parameters of the shower module:

⟨*Shower core: parameters*⟩≡
```
    ! real(default), parameter :: alphasmax = one
    ! real(default), parameter :: xpdfmax = 10._default
    real(default), save :: alphasxpdfmax = 12._default
```

⟨*Shower core: shower: TBP*⟩≡
```
    procedure :: add_interaction_2ton => shower_add_interaction_2ton
```

⟨*Shower core: procedures*⟩≡
```
    subroutine shower_add_interaction_2ton (shower, partons)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t), intent(in), dimension(:), allocatable :: partons
      type(ckkw_pseudo_shower_weights_t) :: ckkw_pseudo_weights
      call shower%add_interaction_2ton_CKKW (partons, ckkw_pseudo_weights)
    end subroutine shower_add_interaction_2ton
```

In this routine, `y` and `ymin` are the jet measures, `w` and `wmax` are weights, `s` is the kinematic energy squared of the interaction. The flag `isr_is_possible` checks whether the initial parton is set, lepton-hadron collisions are not implemented (yet). As a workaround: as WHIZARD can treat partons as massless, there might be partons with $E < m$: if such a parton is found, quarks will be treated massless.

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: add_interaction_2ton_CKKW => shower_add_interaction_2ton_CKKW
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_add_interaction_2ton_CKKW &
        (shower, partons, ckkw_pseudo_weights)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t), intent(in), dimension(:), allocatable :: partons
      type(ckkw_pseudo_shower_weights_t), intent(in) :: ckkw_pseudo_weights

      integer :: n_partons, n_out, n_partons_shower
      integer :: i, j, imin, jmin
      real(default) :: y, ymin
      real(default) :: w, wmax
```

```
real(default) :: random, sum
! real(default) :: s
type(parton_pointer_t), dimension(:), allocatable :: new_partons
type(parton_t), pointer :: prt
integer :: n_int
type(shower_interaction_pointer_t), dimension(:), allocatable :: temp
type(vector4_t) :: prtmomentum, childmomentum
logical :: isr_is_possible
type(lorentz_transformation_t) :: L

if (signal_is_pending ()) return
n_partons = size (partons)
n_out = n_partons - 2
if (n_out < 2) then
   call msg_bug &
        ("Shower core: trying to add a 2-> (something<2) interaction")
end if
! print *, " adding a 2-> ", n_out, " interaction"

isr_is_possible = associated (partons(1)%p%initial) .and. &
     associated (partons(2)%p%initial)

if (associated (partons(1)%p%initial) .and. &
     parton_is_quark (partons(1)%p)) then
   if (parton_get_energy (partons(1)%p) < &
        2._double * parton_mass (partons(1)%p)) then
     if (abs(partons(1)%p%type) < 2) then
        treat_light_quarks_massless = .true.
     else
        treat_duscb_quarks_massless = .true.
     end if
   end if
end if
if (associated (partons(2)%p%initial) .and. &
     parton_is_quark (partons(2)%p)) then
   if (parton_get_energy (partons(2)%p) < &
        two * parton_mass (partons(2)%p)) then
     if (abs(partons(2)%p%type) < 2) then
        treat_light_quarks_massless = .true.
     else
        treat_duscb_quarks_massless = .true.
     end if
   end if
end if
```

⟨*Add a new interaction to* `shower%interactions`⟩

```
if (associated (shower%interactions(n_int)%i%partons(1)%p%initial))  &
     call parton_set_simulated &
     (shower%interactions(n_int)%i%partons(1)%p%initial)
if (associated (shower%interactions(n_int)%i%partons(2)%p%initial))  &
     call parton_set_simulated &
     (shower%interactions(n_int)%i%partons(2)%p%initial)
if (isr_is_possible) then
```

```
   !!! boost to the CMFrame of the incoming partons
   L = boost (-(shower%interactions(n_int)%i%partons(1)%p%momentum + &
         shower%interactions(n_int)%i%partons(2)%p%momentum), &
         (shower%interactions(n_int)%i%partons(1)%p%momentum + &
         shower%interactions(n_int)%i%partons(2)%p%momentum)**1 )
   do i = 1, n_partons
      call parton_apply_lorentztrafo &
            (shower%interactions(n_int)%i%partons(i)%p, L)
   end do
end if
do i = 1, size (partons)
   if (signal_is_pending ()) return
   !!! ensure that partons are marked as belonging to the hard interaction
   shower%interactions(n_int)%i%partons(i)%p%belongstointeraction &
         = .true.
   !!! ensure that incoming partons are marked as belonging to ISR
   shower%interactions(n_int)%i%partons(i)%p%belongstoFSR = i > 2

   shower%interactions(n_int)%i%partons(i)%p%interactionnr &
         = n_int
   !!! include a 2^(i - 1) number as a label for the ckkw clustering
   shower%interactions(n_int)%i%partons(i)%p%ckkwlabel = 2**(i - 1)
end do
```

⟨*Add partons from* `shower%interactions` *to* `shower%partons`⟩

```
if (isr_is_possible) then
   if (isr_pt_ordered) then
      call shower_prepare_for_simulate_isr_pt &
            (shower, shower%interactions(size (shower%interactions))%i)
   else
      call shower_prepare_for_simulate_isr_ana_test &
            (shower, shower%interactions(n_int)%i%partons(1)%p, &
            shower%interactions(n_int)%i%partons(2)%p)
   end if
end if

!!! generate pseudo parton shower history and add all partons to
!!!         shower%partons-array
!!! TODO initial -> initial + final branchings ??
allocate (new_partons(1:(n_partons - 2)))
do i = 1, size (new_partons)
   nullify (new_partons(i)%p)
end do
do i = 1, size (new_partons)
   new_partons(i)%p => shower%interactions(n_int)%i%partons(i + 2)%p
end do
imin = 0
jmin = 0

if (allocated (ckkw_pseudo_weights%weights)) then
   ⟨Perform clustering using the CKKW weights⟩
else
   ⟨Perform clustering in the usual way⟩
```

```
        end if

        !!! add all partons to the shower
        ⟨Count number of associated shower%partons(i)%p⟩

        !!! set the FSR starting scale for all partons
        do i = 1, size (new_partons)
           !!! the imaginary mother is the only parton remaining in new_partons
           if (.not. associated (new_partons(i)%p)) cycle

           call set_starting_scale (new_partons(i)%p, &
                get_starting_scale (new_partons(i)%p))
           exit
        end do

        deallocate (new_partons)

        !!! call shower%write ()
        !!! print *, "end of shower_interactionadd2ton"

     contains

        ⟨Procedures of shower_add_interaction_2ton_CKKW⟩
     end subroutine shower_add_interaction_2ton_CKKW


⟨Add a new interaction to shower%interactions⟩≡
  if (allocated (shower%interactions)) then
     n_int = size (shower%interactions) + 1
  else
     n_int = 1
  end if
  allocate (temp (1:n_int))
  do i = 1, n_int - 1
     allocate (temp(i)%i)
     temp(i)%i = shower%interactions(i)%i
  end do
  allocate (temp(n_int)%i)
  allocate (temp(n_int)%i%partons(1:n_partons))
  do i = 1, n_partons
     allocate (temp(n_int)%i%partons(i)%p)
     call parton_copy (partons(i)%p, temp(n_int)%i%partons(i)%p)
  end do
  if (allocated (shower%interactions)) deallocate(shower%interactions)
  allocate (shower%interactions(1:n_int))
  do i = 1, n_int
     shower%interactions(i)%i => temp(i)%i
  end do
  deallocate (temp)

⟨Add partons from shower%interactions to shower%partons⟩≡
  if (allocated (shower%partons)) then
     allocate (new_partons(1:size(shower%partons) + &
          size(shower%interactions(n_int)%i%partons)))
     do i = 1, size (shower%partons)
```

```
            new_partons(i)%p => shower%partons(i)%p
        end do
        do i = 1, size (shower%interactions(n_int)%i%partons)
            new_partons(size(shower%partons) + i)%p => &
                  shower%interactions(n_int)%i%partons(i)%p
        end do
        deallocate (shower%partons)
    else
        allocate (new_partons(1:size(shower%interactions(n_int)%i%partons)))
        do i = 1, size (partons)
            new_partons(i)%p => shower%interactions(n_int)%i%partons(i)%p
        end do
    end if
    allocate (shower%partons(1:size (new_partons)))
    do i = 1, size (new_partons)
        shower%partons(i)%p => new_partons(i)%p
    end do
    deallocate (new_partons)
    !!! call shower%write ()
⟨Perform clustering using the CKKW weights⟩≡
    CKKW_CLUSTERING: do
        !!! search for the combination with the highest weight
        wmax = zero
        CKKW_OUTER: do i = 1, size (new_partons)
            CKKW_INNER: do j = i + 1, size (new_partons)
                if (.not. associated (new_partons(i)%p)) cycle
                if (.not. associated (new_partons(j)%p)) cycle
                w = ckkw_pseudo_weights%weights(new_partons(i)%p%ckkwlabel + &
                     new_partons(j)%p%ckkwlabel)
                if (w > wmax .or. wmax == zero) then
                    wmax = w
                    imin = i
                    jmin = j
                end if
            end do CKKW_INNER
        end do CKKW_OUTER
        if (wmax > zero) then
            call shower%add_parent (new_partons(imin)%p)
            call parton_set_child (new_partons(imin)%p%parent, &
                 new_partons(jmin)%p, 2)
            call parton_set_parent (new_partons(jmin)%p, &
                 new_partons(imin)%p%parent)
            prt => new_partons(imin)%p%parent
            prt%nr = shower_get_next_free_nr (shower)
            prt%type = 94
            !!! something for internal use needed, 81-100 should be reserved
            !!!    for internal purposes

            prt%momentum = new_partons(imin)%p%momentum + &
                 new_partons(jmin)%p%momentum
            prt%t = prt%momentum**2

            !!! auxilliary values for the ckkw matching
            !!! for now, randomly choose the type of the intermediate
```

```
            prt%ckkwlabel = new_partons(imin)%p%ckkwlabel + &
                new_partons(jmin)%p%ckkwlabel
         sum = zero
         call tao_random_number (random)
         CKKW_TYPE: do i = 0, 4
            if (sum + &
                  ckkw_pseudo_weights%weights_by_type(prt%ckkwlabel, i) > &
                  random * ckkw_pseudo_weights%weights(prt%ckkwlabel) ) then
               prt%ckkwtype = i
               exit ckkw_type
            end if
            sum = sum + &
                  ckkw_pseudo_weights%weights_by_type(prt%ckkwlabel, i)
         end do CKKW_TYPE

         !!! TODO -> calculate costheta and store it for
         !!!         later use in generate_ps

         if (space_part_norm(prt%momentum) > 1E-10_default) then
            prtmomentum = prt%momentum
            childmomentum = prt%child1%momentum
            prtmomentum = boost (-parton_get_beta(prt) / &
                  sqrt (one - &
                  (parton_get_beta(prt))**2), space_part (prt%momentum) / &
                  space_part_norm(prt%momentum)) * prtmomentum
            childmomentum = boost (-parton_get_beta(prt) / &
                  sqrt(one - &
                  (parton_get_beta(prt))**2), space_part (prt%momentum) / &
                  space_part_norm(prt%momentum)) * childmomentum
            prt%costheta = enclosed_angle_ct(prtmomentum, childmomentum)
         else
            prt%costheta = - one
         end if

         prt%belongstointeraction = .true.
         prt%belongstoFSR = &
               new_partons(imin)%p%belongstoFSR .and. &
               new_partons(jmin)%p%belongstoFSR

         nullify (new_partons(imin)%p)
         nullify (new_partons(jmin)%p)
         new_partons(imin)%p => prt
      else
         exit CKKW_CLUSTERING
      end if
   end do CKKW_CLUSTERING
```
⟨*Perform clustering in the usual way*⟩≡
```
  CLUSTERING: do
     !!! search for the partons to be clustered together
     ymin = zero
     OUTER: do i = 1, size (new_partons)
        INNER: do j = i + 1, size (new_partons)
           !!! calculate the jet measure
           if (.not.associated (new_partons(i)%p)) cycle INNER
```

2181

```
            if (.not.associated (new_partons(j)%p)) cycle INNER
            if (.not. shower_clustering_allowed &
                  (shower, new_partons, i,j)) &
                  cycle inner
            !!! Durham jet-measure ! don't care about constants
            y = min (parton_get_energy (new_partons(i)%p), &
                  parton_get_energy (new_partons(j)%p)) * &
                  (one - enclosed_angle_ct &
                  (new_partons(i)%p%momentum, &
                  new_partons(j)%p%momentum))
            if (y < ymin .or. ymin == zero) then
                  ymin = y
                  imin = i
                  jmin = j
            end if
      end do INNER
end do OUTER
if (ymin > zero) then
      call shower%add_parent (new_partons(imin)%p)
      call parton_set_child &
            (new_partons(imin)%p%parent, new_partons(jmin)%p, 2)
      call parton_set_parent &
            (new_partons(jmin)%p, new_partons(imin)%p%parent)
      prt => new_partons(imin)%p%parent
      prt%nr = shower_get_next_free_nr (shower)
      prt%type = 94
      !!! something for internal use needed, 81-100 should be
      !!!     reserved for internal purposes

      prt%momentum = new_partons(imin)%p%momentum + &
            new_partons(jmin)%p%momentum
      prt%t = prt%momentum**2
      !!! TODO -> calculate costheta and store it for
      !!!         later use in generate_ps

      if (space_part_norm(prt%momentum) > 1E-10_default) then
            prtmomentum = prt%momentum
            childmomentum = prt%child1%momentum
            prtmomentum = boost (-parton_get_beta(prt) / sqrt(one - &
                  (parton_get_beta(prt))**2), space_part(prt%momentum) / &
                  space_part_norm(prt%momentum)) * prtmomentum
            childmomentum = boost (-parton_get_beta(prt) / &
                  sqrt(one - &
                  (parton_get_beta(prt))**2), space_part(prt%momentum) / &
                  space_part_norm(prt%momentum)) * childmomentum
            prt%costheta = enclosed_angle_ct(prtmomentum, childmomentum)
      else
            prt%costheta = - one
      end if

      prt%belongstointeraction = .true.
      nullify (new_partons(imin)%p)
      nullify (new_partons(jmin)%p)
      new_partons(imin)%p => prt
```

2182

```
        else
            exit CLUSTERING
        end if
    end do CLUSTERING
```

⟨*Count number of associated* `shower%partons(i)%p`⟩≡
```
  n_partons_shower = 0
  if (allocated (shower%partons)) then
      do i = 1, size (shower%partons)
          if (associated (shower%partons(i)%p)) &
              n_partons_shower = n_partons_shower + 1
      end do
  end if
```

⟨*Procedures of* `shower_add_interaction_2ton_CKKW`⟩≡
```
  function shower_clustering_allowed (shower, partons, i, j) result (allowed)
    type(shower_t), intent(inout) :: shower
    logical :: allowed
    type(parton_pointer_t), intent(in), dimension(:), allocatable :: partons
    integer, intent(in) :: i, j
    !!! TODO implement checking if clustering is allowed, e.g.
    !!!    in e+e- -> qqg don't cluster the quarks together first
    allowed = .true.
  end function shower_clustering_allowed
```

⟨*Procedures of* `shower_add_interaction_2ton_CKKW`⟩+≡
```
  recursive subroutine transfer_pointers (destiny, start, prt)
    type(parton_pointer_t), dimension(:), allocatable :: destiny
    integer, intent(inout) :: start
    type(parton_t), pointer :: prt
    destiny(start)%p => prt
    start = start + 1
    if (associated (prt%child1)) then
        call transfer_pointers (destiny, start, prt%child1)
    end if
    if (associated (prt%child2)) then
        call transfer_pointers (destiny, start, prt%child2)
    end if
  end subroutine transfer_pointers
```

⟨*Procedures of* `shower_add_interaction_2ton_CKKW`⟩+≡
```
  recursive function get_starting_scale (prt) result (scale)
    type(parton_t), pointer :: prt
    real(default) :: scale
    scale = huge (scale)
    if (associated (prt%child1) .and. associated (prt%child2)) then
        scale = min(scale, prt%t)
    end if
    if (associated (prt%child1)) then
        scale = min (scale, get_starting_scale (prt%child1))
    end if
    if (associated (prt%child2)) then
        scale = min (scale, get_starting_scale (prt%child2))
    end if
```

2183

```
      end function get_starting_scale
```

⟨*Procedures of* `shower_add_interaction_2ton_CKKW`⟩+≡

```
  recursive subroutine set_starting_scale (prt, scale)
    type(parton_t), pointer :: prt
    real(default) :: scale
    if (prt%type /= 94) then
        if (scale > D_Min_t + parton_mass_squared (prt)) then
           prt%t = scale
        else
           prt%t = parton_mass_squared (prt)
           call parton_set_simulated (prt)
        end if
    end if
    if (associated (prt%child1)) then
        call set_starting_scale (prt%child1, scale)
    end if
    if (associated (prt%child2)) then
        call set_starting_scale (prt%child2, scale)
    end if
  end subroutine set_starting_scale
```

⟨*Shower core: shower: TBP*⟩+≡

```
    procedure :: simulate_no_isr_shower => shower_simulate_no_isr_shower
```

⟨*Shower core: procedures*⟩+≡

```
    subroutine shower_simulate_no_isr_shower (shower)
      class(shower_t), intent(inout) :: shower
      integer :: i, j
      type(parton_t), pointer :: prt
      do i = 1, size (shower%interactions)
         do j = 1, 2
            prt => shower%interactions(i)%i%partons(j)%p
            if (associated (prt%initial)) then
               !!! for virtuality ordered: remove unneeded partons
               if (associated (prt%parent)) then
                  if (.not. parton_is_hadron(prt%parent)) then
                     if (associated (prt%parent%parent)) then
                        if (.not. parton_is_hadron (prt%parent)) then
                           call shower_remove_parton_from_partons &
                                (shower, prt%parent%parent)
                        end if
                     end if
                     call shower_remove_parton_from_partons &
                          (shower, prt%parent)
                  end if
               end if
               call parton_set_parent (prt, prt%initial)
               call parton_set_child (prt%initial, prt, 1)
               if (associated (prt%initial%child2)) then
                  call shower_remove_parton_from_partons &
                       (shower,prt%initial%child2)
                  deallocate (prt%initial%child2)
               end if
```

```
          call shower%add_child (prt%initial, 2)
        end if
      end do
    end do
  end subroutine shower_simulate_no_isr_shower
```

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: simulate_no_fsr_shower => shower_simulate_no_fsr_shower
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_simulate_no_fsr_shower (shower)
      class(shower_t), intent(inout) :: shower
      integer :: i, j
      type(parton_t), pointer :: prt
      do i = 1, size (shower%interactions)
        do j = 3, size (shower%interactions(i)%i%partons)
          prt => shower%interactions(i)%i%partons(j)%p
          call parton_set_simulated (prt)
          prt%scale = 0._double
          prt%t = parton_mass_squared (prt)
        end do
      end do
    end subroutine shower_simulate_no_fsr_shower
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine swap_pointers (prtp1, prtp2)
      type(parton_pointer_t), intent(inout) :: prtp1, prtp2
      type(parton_pointer_t) :: prtptemp
      prtptemp%p => prtp1%p
      prtp1%p => prtp2%p
      prtp2%p => prtptemp%p
    end subroutine swap_pointers
```

This removes emitted timelike partons.

⟨*Shower core: procedures*⟩+≡
```
    recursive subroutine shower_remove_parton_from_partons (shower, prt)
      type(shower_t), intent(inout) :: shower
      type(parton_t), pointer :: prt
      integer :: i
      if (.not. prt%belongstoFSR .and. associated (prt%child2)) then
        call shower_remove_parton_from_partons_recursive (shower, prt%child2)
      end if
      do i = 1, size (shower%partons)
        if (associated (shower%partons(i)%p, prt)) then
          shower%partons(i)%p => null()
          exit
        end if
        if (i == size (shower%partons)) then
          ! call msg_bug ("Shower core: parton to be removed not found")
        end if
      end do
    end subroutine shower_remove_parton_from_partons
```

This removes the parton `prt` and all its children.

⟨*Shower core: procedures*⟩+≡
```
recursive subroutine shower_remove_parton_from_partons_recursive (shower, prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), pointer :: prt
  if (associated (prt%child1)) then
     call shower_remove_parton_from_partons_recursive (shower, prt%child1)
     deallocate (prt%child1)
  end if
  if (associated (prt%child2)) then
     call shower_remove_parton_from_partons_recursive (shower, prt%child2)
     deallocate (prt%child2)
  end if
  call shower_remove_parton_from_partons (shower, prt)
end subroutine shower_remove_parton_from_partons_recursive
```

⟨*Shower core: shower: TBP*⟩+≡
```
procedure :: sort_partons => shower_sort_partons
```

⟨*Shower core: procedures*⟩+≡
```
subroutine shower_sort_partons (shower)
  class(shower_t), intent(inout) :: shower
  integer :: i, j, maxsort, size_partons
  logical :: changed
  !!! print *, " shower_sort_partons"
  if (.not. allocated (shower%partons)) return
  size_partons = size (shower%partons)
  maxsort = 0
  do i = 1, size_partons
     if (signal_is_pending ()) return
     if (associated (shower%partons(i)%p))  maxsort = i
  end do
  size_partons = size (shower%partons)
  if (size_partons <= 1) return
  do i = 1, maxsort
     if (signal_is_pending ()) return
     if (.not. associated (shower%partons(i)%p)) cycle
     if (.not. isr_pt_ordered) then
        !!! set unsimulated ISR partons to be "typeless" to prevent
        !!!     influences from "wrong" masses
        if (.not. shower%partons(i)%p%belongstoFSR .and. &
             .not. parton_is_simulated (shower%partons(i)%p) .and. &
             .not. shower%partons(i)%p%belongstointeraction) then
           shower%partons(i)%p%type = 0
        end if
     end if
  end do
  !!! Just a Bubblesort
  !!! Different algorithms needed for t-ordered and pt^2-ordered shower
  !!! Pt-ordered:
  if (isr_pt_ordered) then
     OUTERDO_PT: do i = 1, maxsort - 1
        changed = .false.
        INNERDO_PT: do j = 1, maxsort - i
```

2186

```
          if (signal_is_pending ()) return
          if (.not. associated (shower%partons(j + 1)%p)) cycle
          if (.not. associated (shower%partons(j)%p)) then
             !!! change if j + 1 ist assoaciated and j is not
             call swap_pointers (shower%partons(j), shower%partons(j + 1))
             changed = .true.
          else if (shower%partons(j)%p%scale < &
               shower%partons(j + 1)%p%scale) then
             call swap_pointers (shower%partons(j), shower%partons(j + 1))
             changed = .true.
          else if (shower%partons(j)%p%scale == &
               shower%partons(j + 1)%p%scale) then
             if (shower%partons(j)%p%nr > shower%partons(j + 1)%p%nr) then
                call swap_pointers (shower%partons(j), shower%partons(j + 1))
                changed = .true.
             end if
          end if
       end do INNERDO_PT
       if (.not. changed) exit OUTERDO_PT
    end do outerdo_pt
!!! |t|-ordered
else
    OUTERDO_T: do i = 1, maxsort - 1
       changed = .false.
       INNERDO_T: do j = 1, maxsort - i
          if (signal_is_pending ()) return
          if (.not. associated (shower%partons(j + 1)%p)) cycle
          if (.not. associated (shower%partons(j)%p)) then
             !!! change if j+1 ist assoaciated and j isn't
             call swap_pointers (shower%partons(j), shower%partons(j + 1))
             changed = .true.
          else if (.not. shower%partons(j)%p%belongstointeraction .and. &
                shower%partons(j + 1)%p%belongstointeraction) then
             !!! move partons belonging to the interaction to the front
             call swap_pointers (shower%partons(j), shower%partons(j + 1))
             changed = .true.
          else if (.not. shower%partons(j)%p%belongstointeraction .and. &
                .not. shower%partons(j + 1)%p%belongstointeraction ) then
             if (abs (shower%partons(j)%p%t) - parton_mass_squared &
                   (shower%partons(j)%p) < &
                   abs(shower%partons(j + 1)%p%t) - parton_mass_squared &
                   (shower%partons(j + 1)%p)) then
                call swap_pointers (shower%partons(j), shower%partons(j + 1))
                changed = .true.
             else
                if (abs (shower%partons(j)%p%t) - parton_mass_squared &
                      (shower%partons(j)%p) == &
                      abs(shower%partons(j + 1)%p%t) - parton_mass_squared &
                      (shower%partons(j + 1)%p)) then
                   if (shower%partons(j)%p%nr > &
                         shower%partons(j + 1)%p%nr) then
                      call swap_pointers (shower%partons(j), &
                            shower%partons(j + 1))
                      changed = .true.
```

```
                      end if
                    end if
                  end if
                end if
              end do INNERDO_T
              if (.not. changed) exit OUTERDO_T
            end do OUTERDO_T
          end if
          !!! print *, "  shower_sort_partons finished"
      end subroutine shower_sort_partons
```

Creation and finalization.

⟨Shower core: shower: TBP⟩+≡
```
      procedure :: create => shower_create
```

⟨Shower core: procedures⟩+≡
```
      subroutine shower_create (shower)
        class(shower_t), intent(inout) :: shower
        shower%next_free_nr = 1
        shower%next_color_nr = 1
        if (allocated (shower%interactions)) then
           call msg_bug ("Shower: creating new shower while old one " // &
                "still associated (interactions)")
        end if
        if (allocated (shower%partons)) then
           call msg_bug ("Shower: creating new shower while old one " // &
                "still associated (partons)")
        end if
        treat_light_quarks_massless = .true.
        treat_duscb_quarks_massless = .false.
        shower%valid = .true.
      end subroutine shower_create
```

Deallocate the interaction pointers.

⟨Shower core: shower: TBP⟩+≡
```
      procedure :: final => shower_final
```

⟨Shower core: procedures⟩+≡
```
      subroutine shower_final (shower)
        class(shower_t), intent(inout) :: shower
        integer :: i
        if (.not. allocated (shower%interactions))  return
        do i = 1, size (shower%interactions)
           if (allocated (shower%interactions(i)%i%partons)) &
                deallocate (shower%interactions(i)%i%partons)
           deallocate (shower%interactions(i)%i)
        end do
        deallocate (shower%interactions)
        deallocate (shower%partons)
      end subroutine shower_final
```

Bookkeeping functions.

⟨Shower core: shower: TBP⟩+≡
```
      procedure :: get_next_free_nr => shower_get_next_free_nr
```

⟨*Shower core: procedures*⟩+≡
```
function shower_get_next_free_nr (shower) result (next_number)
  class(shower_t), intent(inout) :: shower
  integer :: next_number
  next_number = shower%next_free_nr
  shower%next_free_nr = shower%next_free_nr + 1
end function shower_get_next_free_nr
```

⟨*Shower core: shower: TBP*⟩+≡
```
procedure :: set_next_color_nr => shower_set_next_color_nr
```
⟨*Shower core: procedures*⟩+≡
```
subroutine shower_set_next_color_nr (shower, index)
  class(shower_t), intent(inout) :: shower
  integer, intent(in) :: index
  if (index < shower%next_color_nr) then
     call msg_error ("in shower_set_next_color_nr")
  else
     shower%next_color_nr = max(shower%next_color_nr, index)
  end if
end subroutine shower_set_next_color_nr
```

⟨*Shower core: shower: TBP*⟩+≡
```
procedure :: get_next_color_nr => shower_get_next_color_nr
```
⟨*Shower core: procedures*⟩+≡
```
function shower_get_next_color_nr (shower) result (next_color)
  class(shower_t), intent(inout) :: shower
  integer :: next_color
  next_color = shower%next_color_nr
  shower%next_color_nr = shower%next_color_nr + 1
end function shower_get_next_color_nr
```

⟨*Shower core: procedures*⟩+≡
```
subroutine shower_enlarge_partons_array (shower, length)
  type(shower_t), intent(inout) :: shower
  integer, intent(in) :: length
  integer :: i, oldlength
  type(parton_pointer_t), dimension(:), allocatable :: new_partons
  !!! print *, "shower_enlarge_partons_array ", length
  if (length > 0) then
     if (allocated (shower%partons)) then
        oldlength = size (shower%partons)
        allocate (new_partons(1:oldlength))
        do i = 1, oldlength
           new_partons(i)%p => shower%partons(i)%p
        end do
        deallocate (shower%partons)
     else
        oldlength = 0
     end if
     allocate (shower%partons(1:oldlength + length))
     do i = 1, oldlength
```

```
            shower%partons(i)%p => new_partons(i)%p
         end do
         do i = oldlength + 1, oldlength + length
            shower%partons(i)%p => null()
         end do
      else
         call msg_bug ("Shower: no parton_pointers added in shower%partons")
      end if
      !!! print *, "  shower_enlarge_partons_array finished"
   end subroutine shower_enlarge_partons_array
```

⟨*Shower core: shower: TBP*⟩+≡
```
   procedure :: add_child => shower_add_child
```

⟨*Shower core: procedures*⟩+≡
```
   subroutine shower_add_child (shower, prt, child)
     class(shower_t), intent(inout) :: shower
     type(parton_t), pointer :: prt
     !!! type(parton_t), intent(inout), target :: prt
     integer, intent(in) :: child
     integer :: i, lastfree
     type(parton_pointer_t) :: newprt
     !!! print *, " shower_add_child for parton ", prt%nr
     if (child /= 1 .and. child /= 2) then
        call msg_bug ("Shower: Adding child in nonexisting place")
     end if
     allocate (newprt%p)
     newprt%p%nr = shower%get_next_free_nr ()
     !!! add new parton as child
     if (child == 1) then
        prt%child1 => newprt%p
     else
        prt%child2 => newprt%p
     end if
     newprt%p%parent => prt
     newprt%p%interactionnr = prt%interactionnr
     !!! add new parton to shower%partons list
     if (associated (shower%partons (size(shower%partons))%p)) then
        call shower_enlarge_partons_array (shower, 10)
     end if
     !!! find last free pointer and let it point to the new parton
     lastfree = 0
     do i = size (shower%partons), 1, -1
        if (.not. associated (shower%partons(i)%p)) then
           lastfree = i
        end if
     end do
     if (lastfree == 0) then
        call msg_bug ("Shower: no free pointers found")
     end if
     shower%partons(lastfree)%p => newprt%p
   end subroutine shower_add_child
```

⟨*Shower core: shower: TBP*⟩+≡

```
      procedure :: add_parent => shower_add_parent
⟨Shower core: procedures⟩+≡
    subroutine shower_add_parent (shower, prt)
      class(shower_t), intent(inout) :: shower
      type(parton_t), intent(inout), target :: prt
      integer :: i, lastfree
      type(parton_pointer_t) :: newprt
      !!! print *, " shower_add_parent for parton ", prt%nr
      allocate (newprt%p)
      newprt%p%nr = shower%get_next_free_nr ()
      !!! add new parton as parent
      newprt%p%child1 => prt
      prt%parent => newprt%p
      newprt%p%interactionnr = prt%interactionnr
      !!! add new parton to shower%partons list
      if (.not. allocated (shower%partons)) then
         call shower_enlarge_partons_array (shower, 10)
      else if (associated (shower%partons(size(shower%partons))%p)) then
         call shower_enlarge_partons_array (shower, 10)
      end if
      !!! find last free pointer and let it point to the new parton
      lastfree = 0
      do i = size(shower%partons), 1, -1
         if (.not. associated (shower%partons(i)%p)) then
            lastfree = i
         end if
      end do
      if (lastfree == 0) then
         call msg_bug ("Shower: no free pointers found")
      end if
      shower%partons(lastfree)%p => newprt%p
      ! print *, "  shower_add_parent finished"
    end subroutine shower_add_parent


⟨Shower core: procedures⟩+≡
    function shower_get_total_momentum (shower, c) result (mom)
      type(shower_t), intent(in) :: shower
      integer, intent(in) :: c
      real(default) :: mom
      integer :: i
      mom = zero
      if (.not. allocated (shower%partons)) return
      do i = 1, size (shower%partons)
         if (.not. associated (shower%partons(i)%p)) cycle
         if (parton_is_final (shower%partons(i)%p)) then
            select case (c)
            case (0)
               mom = mom + vector4_get_component(shower%partons(i)%p%momentum, 0)
            case (1)
               mom = mom + vector4_get_component(shower%partons(i)%p%momentum, 1)
            case (2)
               mom = mom + vector4_get_component(shower%partons(i)%p%momentum, 2)
            case (3)
```

```
                mom = mom + vector4_get_component(shower%partons(i)%p%momentum, 3)
            case default
                call msg_error ("Shower: wrong component of 4momentum")
            end select
        end if
      end do
    end function shower_get_total_momentum
```

⟨*Shower core: procedures*⟩+≡
```
    function shower_get_nr_of_partons (shower, mine, include_remnants) &
        result (nr)
      type(shower_t), intent(in) :: shower
      real(default), intent(in), optional :: mine
      logical, intent(in), optional :: include_remnants
      integer :: nr
      integer :: i
      type(parton_t), pointer :: prt
      real(default) :: minenergy
      nr = 0
      if (present (mine)) then
         minenergy = mine
      else
         minenergy = zero
      end if
      do i = 1, size (shower%partons)
         prt => shower%partons(i)%p
         if (.not. associated (prt)) cycle
         if (.not. parton_is_final (prt)) cycle
         if (prt%type == 9999) then
            if (present (include_remnants)) then
               if (.not. include_remnants) cycle
            end if
         end if
         if (present(mine)) then
            if (parton_get_energy (prt) > mine) then
               nr = nr + 1
            end if
         else
            nr = nr +1
         end if
      end do
    end function shower_get_nr_of_partons
```

⟨*Shower core: procedures*⟩+≡
```
    function shower_get_nr_of_final_colored_ME_partons (shower) result(nr)
      type(shower_t), intent(in) :: shower
      integer :: nr
      integer :: i, j
      type(parton_t), pointer :: prt
      nr = 0
      do i = 1, size (shower%interactions)
         do j = 1, size (shower%interactions(i)%i%partons)
            prt => shower%interactions(i)%i%partons(j)%p
```

```
            if (.not. associated (prt)) cycle
            if (.not. parton_is_colored (prt)) cycle
            if (prt%belongstointeraction .and. prt%belongstoFSR .and. &
                (prt%type /= 94)) then
               nr = nr +1
            end if
         end do
      end do
    end function shower_get_nr_of_final_colored_ME_partons
```

⟨Shower core: shower: TBP⟩+≡
```
    procedure :: get_final_colored_ME_partons => &
        shower_get_final_colored_ME_partons
```
⟨Shower core: procedures⟩+≡
```
    subroutine shower_get_final_colored_ME_partons (shower, partons)
      class(shower_t), intent(in) :: shower
      type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: partons
      integer :: i, j, index, s
      type(parton_t), pointer :: prt
      if (allocated(partons))  deallocate(partons)
      s = shower_get_nr_of_final_colored_ME_partons (shower)
      if (s == 0) return
      allocate (partons(1:s))
      index = 0
      do i = 1, size (shower%interactions)
         do j = 1, size (shower%interactions(i)%i%partons)
            prt => shower%interactions(i)%i%partons(j)%p
            if (.not. associated (prt)) cycle
            if (.not. parton_is_colored (prt)) cycle
            if (prt%belongstointeraction .and. prt%belongstoFSR .and. &
                (prt%type /= 94)) then
               index = index + 1
               partons(index)%p => prt
            end if
         end do
      end do
    end subroutine shower_get_final_colored_ME_partons
```

⟨Shower core: public⟩+≡
```
    public :: shower_get_final_partons
```
⟨Shower core: procedures⟩+≡
```
    subroutine shower_get_final_partons (shower, partons, include_remnants)
      type(shower_t), intent(in) :: shower
      type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: partons
      logical, intent(in), optional :: include_remnants
      integer :: i, j
      type(parton_t), pointer :: prt
      if (allocated (partons))  deallocate (partons)
      allocate (partons(1:shower_get_nr_of_partons(shower, &
          include_remnants = include_remnants)))
      j = 0
      do i = 1, size (shower%partons)
```

```
            prt => shower%partons(i)%p
            if (.not. associated (prt)) cycle
            if (.not. parton_is_final (prt)) cycle
            !!! remnant
            if (prt%type == 9999) then
               if (present (include_remnants)) then
                  if (.not. include_remnants) cycle
               end if
            end if
            j = j + 1
            partons(j)%p => prt
         end do
    end subroutine shower_get_final_partons
```

⟨*Shower core: procedures*⟩+≡

```
    recursive function interaction_fsr_is_finished_for_parton(prt) result (finished)
      type(parton_t), intent(in) :: prt
      logical :: finished
      if (prt%belongstoFSR) then
         !!! FSR partons
         if (associated (prt%child1)) then
            finished = interaction_fsr_is_finished_for_parton (prt%child1) &
                  .and. interaction_fsr_is_finished_for_parton (prt%child2)
         else
            finished = prt%t <= parton_mass_squared(prt)
         end if
      else
         !!! search for emitted timelike partons in ISR shower
         if (.not. associated (prt%initial)) then
            !!! no inital -> no ISR
            finished = .true.
         else if (.not. associated (prt%parent)) then
            finished = .false.
         else
            if (.not. parton_is_hadron (prt%parent)) then
               if (associated (prt%child2)) then
                  finished = interaction_fsr_is_finished_for_parton (prt%parent) .and. &
                        interaction_fsr_is_finished_for_parton (prt%child2)
               else
                  finished = interaction_fsr_is_finished_for_parton (prt%parent)
               end if
            else
               if (associated (prt%child2)) then
                  finished = interaction_fsr_is_finished_for_parton (prt%child2)
               else
                  !!! only second partons can come here -> if that happens FSR
                  !!!    evolution is not existing
                  finished = .true.
               end if
            end if
         end if
      end if
    end function interaction_fsr_is_finished_for_parton
```

⟨*Shower core: procedures*⟩+≡
```
function interaction_fsr_is_finished (interaction) result (finished)
  type(shower_interaction_t), intent(in) :: interaction
  logical :: finished
  integer :: i
  finished = .true.
  if (.not. allocated (interaction%partons)) return
  do i = 1, size (interaction%partons)
     if (.not. interaction_fsr_is_finished_for_parton &
          (interaction%partons(i)%p)) then
        finished = .false.
        exit
     end if
  end do
end function interaction_fsr_is_finished
```

⟨*Shower core: public*⟩+≡
```
public :: shower_interaction_get_shat
```
⟨*Shower core: procedures*⟩+≡
```
function shower_interaction_get_shat (interaction) result (shat)
  type(shower_interaction_t), intent(in) :: interaction
  real(default) :: shat
  shat = (interaction%partons(1)%p%momentum + &
          interaction%partons(2)%p%momentum)**2
end function shower_interaction_get_shat
```

⟨*Shower core: public*⟩+≡
```
public :: shower_interaction_get_s
```
⟨*Shower core: procedures*⟩+≡
```
function shower_interaction_get_s (interaction) result (s)
  type(shower_interaction_t), intent(in) :: interaction
  real(default) :: s
  s = (interaction%partons(1)%p%initial%momentum + &
       interaction%partons(2)%p%initial%momentum)**2
end function shower_interaction_get_s
```

⟨*Shower core: procedures*⟩+≡
```
function shower_fsr_is_finished (shower) result (finished)
  type(shower_t), intent(in) :: shower
  logical :: finished
  integer :: i
  finished = .true.
  if (.not.allocated (shower%interactions)) return
  do i = 1, size(shower%interactions)
     if (.not. interaction_fsr_is_finished (shower%interactions(i)%i)) then
        finished = .false.
        exit
     end if
  end do
end function shower_fsr_is_finished
```

⟨*Shower core: procedures*⟩+≡
```
function shower_isr_is_finished (shower) result (finished)
  type(shower_t), intent(in) :: shower
  logical :: finished
  integer :: i
  type(parton_t), pointer :: prt
  finished = .true.
  if (.not.allocated (shower%partons)) return
  do i = 1, size (shower%partons)
     if (.not. associated (shower%partons(i)%p)) cycle
     prt => shower%partons(i)%p
     if (isr_pt_ordered) then
        if (.not. prt%belongstoFSR .and. .not. parton_is_simulated (prt) &
           .and. prt%scale > zero) then
           finished = .false.
           exit
        end if
     else
        if (.not. prt%belongstoFSR .and. .not. parton_is_simulated (prt) &
           .and. prt%t < zero) then
           finished = .false.
           exit
        end if
     end if
  end do
end function shower_isr_is_finished
```

⟨*Shower core: procedures*⟩+≡
```
function shower_is_finished (shower) result (finished)
  type(shower_t), intent(in) :: shower
  logical :: finished
  finished = shower_isr_is_finished (shower) .and. &
             shower_fsr_is_finished(shower)
end function shower_is_finished
```

⟨*Shower core: procedures*⟩+≡
```
subroutine interaction_find_partons_nearest_to_hadron (interaction, prt1, prt2)
  type(shower_interaction_t), intent(in) :: interaction
  type(parton_t), pointer :: prt1, prt2
  prt1 => null ()
  prt2 => null ()
  prt1 => interaction%partons(1)%p
  do
     if (associated (prt1%parent)) then
        if (parton_is_hadron (prt1%parent)) then
           exit
        else if ((.not. isr_pt_ordered .and. .not. parton_is_simulated (prt1%parent)) &
              .or. (isr_pt_ordered .and. .not. parton_is_simulated (prt1))) then
           exit
        else
           prt1 => prt1%parent
        end if
     else
```

2196

```
            exit
         end if
      end do
      prt2 => interaction%partons(2)%p
      do
         if (associated (prt2%parent)) then
            if (parton_is_hadron (prt2%parent)) then
               exit
            else if ((.not. isr_pt_ordered .and. .not. parton_is_simulated (prt2%parent)) &
                 .or. (isr_pt_ordered .and. .not. parton_is_simulated (prt2))) then
               exit
            else
               prt2 => prt2%parent
            end if
         else
            exit
         end if
      end do
   end subroutine interaction_find_partons_nearest_to_hadron
```

⟨*Shower core: shower: TBP*⟩+≡
```
      procedure :: update_beamremnants => shower_update_beamremnants
```

⟨*Shower core: procedures*⟩+≡
```
   subroutine shower_update_beamremnants (shower)
     class(shower_t), intent(inout) :: shower
     type(parton_t), pointer :: hadron, remnant
     integer :: i
     real(default) :: random
     !!! only proton in first interaction !!?
     !!! currently only first beam-remnant will be updated
     do i = 1,2
        if (associated (shower%interactions(1)%i%partons(i)%p%initial)) then
           hadron => shower%interactions(1)%i%partons(i)%p%initial
        else
           cycle
        end if
        remnant => hadron%child2
        if (associated (remnant)) then
           remnant%momentum = hadron%momentum - hadron%child1%momentum
        end if
        !!! generate flavor of the beam-remnant if beam was proton
        if (abs (hadron%type) == 2212 .and. associated (hadron%child1)) then
           if (parton_is_quark (hadron%child1)) then
              !!! decide if valence (u,d) or sea quark (s,c,b)
              if ((abs (hadron%child1%type) <= 2) .and. &
                   (hadron%type * hadron%child1%type > zero)) then
                 !!! valence quark
                 if (abs (hadron%child1%type) == 1) then
                    !!! if d then remaining diquark is uu_1
                    remnant%type = sign (2203, hadron%type)
                 else
                    call tao_random_number (random)
                    !!! if u then remaining diqaurk is either
```

```
          if (random < 0.75_default) then
             !!! with 75% a ud_0
             remnant%type = sign (2101, hadron%type)
          else
             !!! with 25% a ud_1
             remnant%type = sign (2103, hadron%type)
          end if
       end if
       remnant%c1 = hadron%child1%c2
       remnant%c2 = hadron%child1%c1
    else if ((hadron%type * hadron%child1%type) < zero) then
       !!! antiquark
       if (.not. associated (remnant%child1)) then
          call shower%add_child (remnant, 1)
       end if
       if (.not. associated (remnant%child2)) then
          call shower%add_child (remnant, 2)
       end if
       call tao_random_number (random)
       if (random < 0.6666_default) then
          !!! 2/3 into udq + u
          if (abs (hadron%child1%type) == 1) then
             remnant%child1%type = sign (2112, hadron%type)
          else if (abs (hadron%child1%type) == 2) then
             remnant%child1%type = sign (2212, hadron%type)
          else if (abs (hadron%child1%type) == 3) then
             remnant%child1%type = sign(3212, hadron%type)
          else if (abs (hadron%child1%type) == 4) then
             remnant%child1%type = sign (4212, hadron%type)
          else if (abs (hadron%child1%type) == 5) then
             remnant%child1%type = sign (5212, hadron%type)
          end if
          remnant%child2%type = sign (2, hadron%type)
       else
          !!! 1/3 into uuq + d
          if (abs (hadron%child1%type).eq.1) then
             remnant%child1%type = sign (2212, hadron%type)
          else if (abs (hadron%child1%type).eq.2) then
             remnant%child1%type = sign (2224, hadron%type)
          else if (abs (hadron%child1%type).eq.3) then
             remnant%child1%type = sign (3222, hadron%type)
          else if (abs (hadron%child1%type).eq.4) then
             remnant%child1%type = sign (4222, hadron%type)
          else if (abs (hadron%child1%type).eq.5) then
             remnant%child1%type = sign (5222, hadron%type)
          end if
          remnant%child2%type = sign (1, hadron%type)
       end if
       remnant%c1 = hadron%child1%c2
       remnant%c2 = hadron%child1%c1
       remnant%child1%c1 = 0
       remnant%child1%c2 = 0
       remnant%child2%c1 = remnant%c1
       remnant%child2%c2 = remnant%c2
```

```
              else
                 !!! sea quark
                 if (.not. associated (remnant%child1)) then
                    call shower%add_child (remnant, 1)
                 end if
                 if (.not. associated (remnant%child2)) then
                    call shower%add_child (remnant, 2)
                 end if
                 call tao_random_number (random)
                 if (random < 0.5_default) then
                    !!! 1/2 into usbar + ud_0
                    if (abs (hadron%child1%type) == 3) then
                       remnant%child1%type = sign (321, hadron%type)
                    else if (abs (hadron%child1%type) == 4) then
                       remnant%child1%type = sign (421, hadron%type)
                    else if (abs (hadron%child1%type) == 5) then
                       remnant%child1%type = sign (521, hadron%type)
                    end if
                    remnant%child2%type = sign (2101, hadron%type)
                 else if (random < 0.6666_default) then
                    !!! 1/6 into usbar + ud_1
                    if (abs (hadron%child1%type) == 3) then
                       remnant%child1%type = sign (321, hadron%type)
                    else if (abs (hadron%child1%type) == 4) then
                       remnant%child1%type = sign (421, hadron%type)
                    else if (abs (hadron%child1%type) == 5) then
                       remnant%child1%type = sign (521, hadron%type)
                    end if
                    remnant%child2%type = sign (2103, hadron%type)
                 else
                    !!! 1/3 into dsbar + uu_1
                    if (abs (hadron%child1%type) == 3) then
                       remnant%child1%type = sign (311, hadron%type)
                    else if (abs (hadron%child1%type) == 4) then
                       remnant%child1%type = sign (411, hadron%type)
                    else if (abs (hadron%child1%type) == 5) then
                       remnant%child1%type = sign (511, hadron%type)
                    end if
                    remnant%child2%type = sign (2203, hadron%type)
                 end if
                 remnant%c1 = hadron%child1%c2
                 remnant%c2 = hadron%child1%c1
                 remnant%child2%c1 = remnant%c1
                 remnant%child2%c2 = remnant%c2
              end if
           else if (parton_is_gluon (hadron%child1)) then
              if (.not.associated (remnant%child1)) then
                 call shower%add_child (remnant, 1)
              end if
              if (.not.associated (remnant%child2)) then
                 call shower%add_child (remnant, 2)
              end if
              call tao_random_number (random)
              if (random < 0.5_default) then
```

```
                                !!! 1/2 into u + ud_0
                                remnant%child1%type = sign (2, hadron%type)
                                remnant%child2%type = sign (2101, hadron%type)
                          else if (random < 0.6666_default) then
                                !!! 1/6 into u + ud_1
                                remnant%child1%type = sign (2, hadron%type)
                                remnant%child2%type = sign (2103, hadron%type)
                          else
                                !!! 1/3 into d + uu_1
                                remnant%child1%type = sign (1, hadron%type)
                                remnant%child2%type = sign (2203, hadron%type)
                          end if
                          remnant%c1 = hadron%child1%c2
                          remnant%c2 = hadron%child1%c1
                          if (sign (1,hadron%type) > 0) then
                                remnant%child1%c1 = remnant%c1
                                remnant%child2%c2 = remnant%c2
                          else
                                remnant%child1%c2 = remnant%c2
                                remnant%child2%c1 = remnant%c1
                          end if
                    end if
                    if (associated (remnant%child1)) then
                          !!! don't care about on-shellness for now
                          remnant%child1%momentum = 0.5_default * remnant%momentum
                          remnant%child2%momentum = 0.5_default * remnant%momentum
                          !!! but care about on-shellness for baryons
                          if (mod (remnant%child1%type, 100) >= 10) then
                                !!! check if the third quark is set -> meson or baryon
                                remnant%child1%t = parton_mass_squared(remnant%child1)
                                remnant%child1%momentum = vector4_moving( &
                                      energy(remnant%child1%momentum), &
                                      ( space_part(remnant%child1%momentum) / &
                                      space_part_norm(remnant%child1%momentum) ) *sqrt( &
                                      energy(remnant%child1%momentum)**2 - remnant%child1%t))
                                remnant%child2%momentum = remnant%momentum &
                                      - remnant%child1%momentum
                          end if
                    end if
              end if
        end do
  end subroutine shower_update_beamremnants
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine interaction_apply_lorentztrafo (interaction, L)
    type(shower_interaction_t), intent(inout) :: interaction
    type(lorentz_transformation_t), intent(in) :: L
    type(parton_t), pointer :: prt
    integer :: i
    !!! ISR part
    do i = 1,2
        prt => interaction%partons(i)%p
        !!! loop over ancestors
        MOTHERS: do
```

```
                        !!! boost parton
                        call parton_apply_lorentztrafo (prt, L)
                        if (associated (prt%child2)) then
                            !!! boost emitted timelike parton (and daughters)
                            call parton_apply_lorentztrafo_recursive (prt%child2, L)
                        end if
                        if (associated (prt%parent)) then
                            if (.not. parton_is_hadron (prt%parent)) then
                                prt => prt%parent
                            else
                                exit
                            end if
                        else
                            exit
                        end if
                    end do MOTHERS
                end do
                !!! FSR part
                if (associated (interaction%partons(3)%p%parent)) then
                    !!! pseudo Parton-Shower histora has been generated -> find
                    !!!     mother and go on from there recursively
                    prt => interaction%partons(3)%p
                    do while (associated (prt%parent))
                        prt => prt%parent
                    end do
                    call parton_apply_lorentztrafo_recursive (prt, L)
                else
                    do i = 3, size (interaction%partons)
                        call parton_apply_lorentztrafo (interaction%partons(i)%p, L)
                    end do
                end if
                !!! print *, " end interaction_apply_lorentztrafo"
            end subroutine interaction_apply_lorentztrafo
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_apply_lorentztrafo (shower, L)
      type(shower_t), intent(inout) :: shower
      type(lorentz_transformation_t), intent(in) :: L
      integer :: i
      do i = 1, size (shower%interactions)
          call interaction_apply_lorentztrafo (shower%interactions(i)%i, L)
      end do
    end subroutine shower_apply_lorentztrafo
```

This boosts partons belonging to the interaction to the center-of-mass frame of its partons nearest to the hadron.

⟨*Shower core: procedures*⟩+≡
```
    subroutine interaction_boost_to_CMframe (interaction)
      type(shower_interaction_t), intent(inout) :: interaction
      type(vector4_t) :: beta
      type(parton_t), pointer :: prt1, prt2
      call interaction_find_partons_nearest_to_hadron (interaction, prt1, prt2)
      beta = prt1%momentum + prt2%momentum
```

```
      beta = beta / vector4_get_component(beta,0)
      if (beta**2 > one) then
          call msg_error ("Shower, boost to CM frame: beta > 1")
          return
      end if
      if (space_part(beta)**2 > 1E-14_default) then
          call interaction_apply_lorentztrafo(interaction, &
              boost(space_part(beta)**1 / &
                  sqrt (one - space_part(beta)**2), -direction(beta)))
      end if
  end subroutine interaction_boost_to_CMframe
```

This boosts every interaction to the center-of-mass-frame of its partons nearest to the hadron.

⟨*Shower core: procedures*⟩+≡
```
  subroutine shower_boost_to_CMframe (shower)
    type(shower_t), intent(inout) :: shower
    integer :: i
    do i = 1, size (shower%interactions)
        call interaction_boost_to_CMframe (shower%interactions(i)%i)
    end do
    call shower%update_beamremnants ()
  end subroutine shower_boost_to_CMframe
```

This boost all partons so that initial partons have their assigned x-value.

⟨*Shower core: shower: TBP*⟩+≡
```
  procedure :: boost_to_labframe => shower_boost_to_labframe
```

⟨*Shower core: procedures*⟩+≡
```
  subroutine shower_boost_to_labframe (shower)
    class(shower_t), intent(inout) :: shower
    integer :: i
    do i = 1, size (shower%interactions)
        call interaction_boost_to_labframe (shower%interactions(i)%i)
    end do
  end subroutine shower_boost_to_labframe
```

This boosts all partons so that initial partons have their assigned x-value.

⟨*Shower core: procedures*⟩+≡
```
  subroutine interaction_boost_to_labframe (interaction)
    type(shower_interaction_t), intent(inout) :: interaction
    type(parton_t), pointer :: prt1, prt2
    type(vector3_t) :: beta
    call interaction_find_partons_nearest_to_hadron (interaction, prt1, prt2)
    if (.not. associated (prt1%initial) .or. .not. associated (prt2%initial)) then
        return
    end if
    !!! if no branching at all occured, take original partons
    ! if (prt1%child1%belongstointeraction .and. prt2%child1%belongstointeraction) then
    !     prt1 => prt1%child1
    !     prt2 => prt2%child1
    ! end if
    !!! transform partons to overall labframe.
```

```
      beta = vector3_canonical(3) * &
          ((prt1%x * vector4_get_component (prt2%momentum, 0) - &
            prt2%x * vector4_get_component (prt1%momentum, 0)) / &
           (prt1%x * vector4_get_component (prt2%momentum, 3) - &
            prt2%x * vector4_get_component (prt1%momentum, 3)))
      if (beta**1 > 1E-10_default) &
          call interaction_apply_lorentztrafo (interaction, &
              boost (beta**1 / sqrt(one - beta**2), -direction(beta)))
    end subroutine interaction_boost_to_labframe
```

Only rotate to z if inital hadrons are given (and they are assumed to be aligned along the z-axis).

⟨*Shower core: procedures*⟩+≡
```
    subroutine interaction_rotate_to_z (interaction)
      type(shower_interaction_t), intent(inout) :: interaction
      type(parton_t), pointer :: prt1, prt2
      call interaction_find_partons_nearest_to_hadron (interaction, prt1, prt2)
      if (associated (prt1%initial)) then
          call interaction_apply_lorentztrafo (interaction, &
              rotation_to_2nd (space_part (prt1%momentum), &
              vector3_canonical(3) * sign (one, &
              vector4_get_component(prt1%initial%momentum,3))))
      end if
    end subroutine interaction_rotate_to_z
```

Rotate initial partons to lie along ±z axis.

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_rotate_to_z (shower)
      type(shower_t), intent(inout) :: shower
      integer :: i
      do i = 1, size (shower%interactions)
          call interaction_rotate_to_z (shower%interactions(i)%i)
      end do
      call shower%update_beamremnants ()
    end subroutine shower_rotate_to_z
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine interaction_generate_primordial_kt (interaction)
      type(shower_interaction_t), intent(inout) :: interaction
      type(parton_t), pointer :: had1, had2
      type(vector4_t) :: momenta(2)
      type(vector3_t) :: beta
      real(default) :: pt (2), phi(2)
      real(default) :: shat
      !!! variables for boosting and rotating
      real(default) :: btheta, bphi
      integer :: i
      if (primordial_kt_width == zero)  return
      !!! print *, "interaction_generate_primordial_kt"
      !!! Return if there are no initials, electron-hadron collision not implemented
      if (.not. associated (interaction%partons(1)%p%initial) .or. &
          .not. associated (interaction%partons(2)%p%initial)) then
```

```
            return
         end if
         had1 => interaction%partons(1)%p%initial
         had2 => interaction%partons(2)%p%initial
         !!! copy momenta and energy
         momenta(1) = had1%child1%momentum
         momenta(2) = had2%child1%momentum
         GENERATE_PT_PHI: do i = 1, 2
            !!! generate transverse momentum and phi
            GENERATE_PT: do
               call tao_random_number (pt (i))
               pt(i) = primordial_kt_width * sqrt(-log(pt(i)))
               if (pt(i) < primordial_kt_cutoff) exit
            end do GENERATE_PT
            call tao_random_number (phi (i))
            phi(i) = twopi * phi(i)
         end do GENERATE_PT_PHI
         !!! adjust momenta
         shat = (momenta(1) + momenta(2))**2
         momenta(1) = vector4_moving (vector4_get_component (momenta(1),0), &
              vector3_moving ([pt(1) * cos(phi(1)), pt(1) * sin(phi(1)), &
              vector4_get_component (momenta(1), 3)]))
         momenta(2) = vector4_moving (vector4_get_component (momenta(2),0), &
              vector3_moving ([pt(2) * cos(phi(2)), pt(2) * sin(phi(2)), &
              vector4_get_component (momenta(2), 3)]))
         beta = vector3_moving ([vector4_get_component (momenta(1), 1) + &
              vector4_get_component (momenta(2), 1) , &
              vector4_get_component (momenta(1), 2) + &
              vector4_get_component (momenta(2), 2), zero]) / sqrt(shat)
         momenta(1) = boost (beta**1 / sqrt(one - beta**2), -direction(beta)) * momenta(1)
         bphi = azimuthal_angle (momenta(1))
         btheta = polar_angle (momenta(1))
         call interaction_apply_lorentztrafo (interaction, &
              rotation (cos(bphi), sin(bphi), 3) * rotation(cos(btheta), &
              sin(btheta), 2) * rotation(cos(-bphi), sin(-bphi), 3))
         call interaction_apply_lorentztrafo (interaction, &
              boost (beta**1 / sqrt(one - beta**2), -direction(beta)))
      end subroutine interaction_generate_primordial_kt
```

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: generate_primordial_kt => shower_generate_primordial_kt
```
⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_generate_primordial_kt (shower)
      class(shower_t), intent(inout) :: shower
      integer :: i
      ! print *, "shower_generate_primordial_kt"
      ! call shower%write ()
      do i = 1, size (shower%interactions)
         call interaction_generate_primordial_kt (shower%interactions(i)%i)
      end do
      call shower%update_beamremnants ()
      ! call shower%write ()
    end subroutine shower_generate_primordial_kt
```

Output.

⟨*Shower core: procedures*⟩+≡
```
    subroutine interaction_write (interaction, unit)
      type(shower_interaction_t), intent(in) :: interaction
      integer, intent(in), optional :: unit
      integer :: i, u
      u = output_unit (unit); if (u < 0) return
      if (associated (interaction%partons(1)%p)) then
         if (associated (interaction%partons(1)%p%initial)) &
              call parton_write (interaction%partons(1)%p%initial, u)
      end if
      if (associated (interaction%partons(2)%p)) then
         if (associated (interaction%partons(2)%p%initial)) &
              call parton_write (interaction%partons(2)%p%initial, u)
      end if
      if (allocated (interaction%partons)) then
         do i = 1, size (interaction%partons)
            call parton_write (interaction%partons(i)%p, u)
         end do
      end if
      write (u, "(A)")
    end subroutine interaction_write
```

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: write => shower_write
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_write (shower, unit)
      class(shower_t), intent(in) :: shower
      integer, intent(in), optional :: unit
      integer :: i, u
      u = output_unit (unit); if (u < 0) return
      if (size (shower%interactions) > 0) then
         write (u, "(3x,A)") "Interactions: "
         do i = 1, size (shower%interactions)
            write (u, "(4x,A,I0)") "Interaction number ", i
            if (.not. associated (shower%interactions(i)%i)) then
               call msg_fatal ("Shower: missing interaction in shower")
            end if
            call interaction_write (shower%interactions(i)%i, u)
         end do
      else
         write (u, "(3x,A)") "[no interactions in shower]"
      end if
      write (u, "(A)")
      if (allocated (shower%partons)) then
         write (u, "(5x,A)")  "Partons:"
         do i = 1, size (shower%partons)
            ! print *, " i = ", i
            if (associated (shower%partons(i)%p)) then
               call parton_write(shower%partons(i)%p, u)
               if (i < size (shower%partons)) then
```

2205

```
                    if (associated (shower%partons(i + 1)%p)) then
                        if (shower%partons(i)%p%belongstointeraction .and. &
                             .not. shower%partons(i + 1)%p%belongstointeraction) then
                            write (u, "(A)") &
                               "----------------------------------------------------"
                            ! stop "END"
                        end if
                    end if
                end if
            end if
        end do
    else
        write (u, "(5x,A)")  "[no partons in shower]"
    end if
    write (u, "(4x,A,4(ES12.5,A))")  "Total Momentum [0:3]: ", &
         shower_get_total_momentum (shower, 0), TAB, &
         shower_get_total_momentum (shower, 1), TAB, &
         shower_get_total_momentum (shower, 2), TAB, &
         shower_get_total_momentum (shower, 3), TAB
    write (u, "(1x,A,L1)") "ISR finished: ", shower_isr_is_finished (shower)
    write (u, "(1x,A,L1)") "FSR finished: ", shower_fsr_is_finished (shower)
  end subroutine shower_write
```

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: write_lhef => shower_write_lhef
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_write_lhef (shower, unit)
      class(shower_t), intent(in) :: shower
      integer, intent(in), optional :: unit
      integer :: u
      integer :: i
      integer :: c1, c2
      u = output_unit (unit);  if (u < 0)  return
      write(u,'(A)') '<LesHouchesEvents version="1.0">'
      write(u,'(A)') '<-- not a complete lhe file - just one event -->'
      write(u,'(A)') '<event>'
      write(u, *) 2 + shower_get_nr_of_partons (shower), 1, 1.0, 1.0, 1.0, 1.0
      !!! write incoming partons
      do i = 1, 2
         if (abs (shower%partons(i)%p%type) < 1000) then
            c1 = 0
            c2 = 0
            if (parton_is_colored (shower%partons(i)%p)) then
               if (shower%partons(i)%p%c1 /= 0) c1 = 500 + shower%partons(i)%p%c1
               if (shower%partons(i)%p%c2 /= 0) c2 = 500 + shower%partons(i)%p%c2
            end if
            write(u,*) shower%partons(i)%p%type, -1, 0, 0, c1, c2, &
                 vector4_get_component (shower%partons(i)%p%momentum, 1), &
                 vector4_get_component (shower%partons(i)%p%momentum, 2), &
                 vector4_get_component (shower%partons(i)%p%momentum, 3), &
                 vector4_get_component (shower%partons(i)%p%momentum, 0), &
                 shower%partons(i)%p%momentum**2, zero, 9.0
         else
```

```
            write(u,*) shower%partons(i)%p%type, -9, 0, 0, 0, 0, &
                 vector4_get_component (shower%partons(i)%p%momentum, 1), &
                 vector4_get_component (shower%partons(i)%p%momentum, 2), &
                 vector4_get_component (shower%partons(i)%p%momentum, 3), &
                 vector4_get_component (shower%partons(i)%p%momentum, 0), &
                 shower%partons(i)%p%momentum**2, zero, 9.0
         end if
      end do
      !!! write outgoing partons
      do i = 3, size (shower%partons)
         if (.not. associated (shower%partons(i)%p)) cycle
         if (.not. parton_is_final (shower%partons(i)%p)) cycle
         c1 = 0
         c2 = 0
         if (parton_is_colored (shower%partons(i)%p)) then
            if (shower%partons(i)%p%c1 .ne. 0) c1 = 500 + shower%partons(i)%p%c1
            if (shower%partons(i)%p%c2 .ne. 0) c2 = 500 + shower%partons(i)%p%c2
         end if
         write(u,*) shower%partons(i)%p%type, 1, 1, 2, c1, c2, &
                 vector4_get_component (shower%partons(i)%p%momentum, 1), &
                 vector4_get_component (shower%partons(i)%p%momentum, 2), &
                 vector4_get_component (shower%partons(i)%p%momentum, 3), &
                 vector4_get_component (shower%partons(i)%p%momentum, 0), &
                 shower%partons(i)%p%momentum**2, zero, 9.0
      end do
      write(u,'(A)') '</event>'
      write(u,'(A)') '</LesHouchesEvents>'
   end subroutine shower_write_lhef
```

⟨*Shower core: procedures*⟩+≡
```
   subroutine shower_replace_parent_by_hadron (shower, prt)
     type(shower_t), intent(inout) :: shower
     type(parton_t), intent(inout), target :: prt
     type(parton_t), pointer :: remnant => null()
     ! print *, " shower_replace_parent_by_hadron for parton ", prt%nr
     if (associated (prt%parent)) then
        call shower_remove_parton_from_partons (shower, prt%parent)
        deallocate (prt%parent)
     end if
     if (.not. associated (prt%initial%child2)) then
        call shower%add_child (prt%initial, 2)
     end if
     prt%parent => prt%initial
     prt%parent%child1 => prt
     ! make other child to be a beam-remnant
     remnant => prt%initial%child2
     remnant%type = 9999
     remnant%momentum = prt%parent%momentum - prt%momentum
     remnant%x = one - prt%x
     remnant%parent => prt%initial
     remnant%t = zero
   end subroutine shower_replace_parent_by_hadron
```

2207

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_get_first_ISR_scale_for_parton (shower, prt, tmax)
  type(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout), target :: prt
  real(default), intent(in), optional :: tmax
  real(default) :: t, tstep, random, integral, temp1
  real(default) :: temprand
  if (present(tmax)) then
     t = max (max (-tscalefactor_isr * parton_get_energy (prt)**2, &
          -abs(tmax)), prt%t)
  else
     t = max (-tscalefactor_isr * parton_get_energy (prt)**2, prt%t)
  end if
  call tao_random_number (random)
  random = -twopi * log(random)
  !!! compare Integral and log(random) instead of random and exp(-Integral)
  random = random / first_integral_suppression_factor
  integral = zero
  call parton_set_simulated (prt, .false.)
  do
     call tao_random_number (temprand)
     tstep = max (abs (0.01_default * t) * temprand, 0.1_default * D_Min_t)
     if (t + 0.5_default * tstep > - D_Min_t) then
        prt%t = parton_mass_squared (prt)
        call parton_set_simulated (prt)
        exit
     end if
     prt%t = t + 0.5_default * tstep
     temp1 = integral_over_z_simple (prt, (random - integral) / tstep)
     integral = integral + tstep * temp1
     if (integral > random) then
        prt%t = t + 0.5_default * tstep
        exit
     end if
     t = t + tstep
  end do
  if (prt%t > - D_Min_t) then
     call shower_replace_parent_by_hadron (shower, prt)
  end if
  ! print *, "  shower_get_first_ISR_scale_for_parton finished"

contains

  function integral_over_z_simple (prt, final) result (integral)
    type(parton_t), intent(inout) :: prt
    ! real(default), intent(in) :: shat, s,
    real(default), intent(in) :: final
    real(default) :: integral

    real(default), parameter :: zstepfactor = one
    real(default), parameter :: zstepmin = 0.0001_default
    real(default) :: z, zstep, minz, maxz
    real(default) :: pdfsum
    integer :: quark
```

2208

```
integral = zero
if (D_print) then
   print *, "integral_over_z_simple for t = ", prt%t
end if
minz = prt%x
! maxz = maxzz(shat, s)
maxz = maxz_isr
z = minz
!!! TODO -> Adapt zstep to structure of divergencies
if (parton_is_gluon (prt%child1)) then
   !!! gluon coming from g->gg
   do
      call tao_random_number (temprand)
      zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
      zstep = min(zstep, maxz - z)
      integral = integral + zstep * (D_alpha_s_isr ((one - &
         (z + 0.5_default * zstep)) * abs(prt%t)) / (abs(prt%t)) * &
         P_ggg (z + 0.5_default * zstep) * get_pdf (prt%initial%type, &
         prt%x / (z + 0.5_default * zstep), abs(prt%t), 21)
      if (integral > final) then
         exit
      end if
      z = z + zstep
      if (z >= maxz) then
         exit
      end if
   end do
   !!! gluon coming from q->qg  ! correctly implemented yet?
   if (integral < final) then
      z = minz
      do
         call tao_random_number (temprand)
         zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
         zstep = min(zstep, maxz - z)
         pdfsum = zero
         do quark = -D_Nf, D_Nf
            if (quark == 0) cycle
            pdfsum = pdfsum + get_pdf (prt%initial%type, &
                  prt%x / (z + 0.5_default * zstep), abs(prt%t), quark)
         end do
         integral = integral + zstep * (D_alpha_s_isr &
            ((z + 0.5_default * zstep) * abs(prt%t)) / (abs(prt%t))) * &
               P_qqg (one - (z + 0.5_default * zstep)) * pdfsum
         if (integral > final) then
            exit
         end if
         z = z + zstep
         if (z >= maxz) then
            exit
         end if
      end do
   end if
else if (parton_is_quark (prt%child1)) then
```

```
            !!! quark coming from q->qg
            do
                call tao_random_number(temprand)
                zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
                zstep = min(zstep, maxz - z)
                integral = integral + zstep * (D_alpha_s_isr ((one - &
                    (z + 0.5_default * zstep)) * abs(prt%t)) / (abs(prt%t))) * &
                    P_qqg (z + 0.5_default * zstep) * get_pdf (prt%initial%type, &
                    prt%x / (z + 0.5_default * zstep), abs(prt%t), prt%type)
                if (integral > final) then
                    exit
                end if
                z = z + zstep
                if (z >= maxz) then
                    exit
                end if
            end do
            !!! quark coming from g->qqbar
            if (integral < final) then
                z = minz
                do
                    call tao_random_number (temprand)
                    zstep = max(zstepmin, temprand * zstepfactor * z*(one - z))
                    zstep = min(zstep, maxz - z)
                    integral = integral + zstep * (D_alpha_s_isr &
                        ((one - (z + 0.5_default * zstep)) * abs(prt%t)) / (abs(prt%t))) * &
                        P_gqq (z + 0.5_default * zstep) * get_pdf (prt%initial%type, &
                        prt%x / (z + 0.5_default * zstep), abs(prt%t), 21)
                    if (integral > final) then
                        exit
                    end if
                    z = z + zstep
                    if (z >= maxz) then
                        exit
                    end if
                end do
            end if

        end if
        integral = integral / get_pdf (prt%initial%type, prt%x, &
            abs(prt%t), prt%type)
    end function integral_over_z_simple

  end subroutine shower_get_first_ISR_scale_for_parton
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_prepare_for_simulate_isr_pt (shower, interaction)
      type(shower_t), intent(inout) :: shower
      type(shower_interaction_t), intent(inout) :: interaction
      real(default) :: s
      ! print *, " shower_prepare_for_simulate_isr_pt"
      !!! get sqrts of interaction
      s = (interaction%partons(1)%p%momentum + &
          interaction%partons(2)%p%momentum)**2
```

```
          interaction%partons(1)%p%scale = tscalefactor_isr * 0.25_default * s
          interaction%partons(2)%p%scale = tscalefactor_isr * 0.25_default * s
          ! call shower%add_parent (interaction%partons(1)%p)
          ! call shower$add_parent (interaction%partons(2)%p)
          !
          ! interaction%partons(1)%p%parent%scale = 0.5_default * sqrts
          ! interaction%partons(1)%p%parent%momentum = &
          !     interaction%partons(1)%p%momentum
          ! interaction%partons(1)%p%parent%belongstoFSR = .false.
          ! interaction%partons(1)%p%parent%initial => &
          !     interaction%partons(1)%p%initial
          ! interaction%partons(2)%p%parent%scale = 0.5_default * sqrts
          ! interaction%partons(2)%p%parent%momentum = &
          !      interaction%partons(2)%p%momentum
          ! interaction%partons(2)%p%parent%belongstoFSR = .false.
          ! interaction%partons(2)%p%parent%initial => &
          !     interaction%partons(2)%p%initial
          !
          ! call shower%add_child (interaction%partons(1)%p%parent, 2)
          ! call shower%add_child (interaction%partons(2)%p%parent, 2)
          ! print *, " shower_prepare_for_simulate_isr_pt finished"
        end subroutine shower_prepare_for_simulate_isr_pt

⟨Shower core: procedures⟩+≡
        subroutine shower_prepare_for_simulate_isr_ana_test (shower, prt1, prt2)
          type(shower_t), intent(inout) :: shower
          type(parton_t), intent(inout), target :: prt1, prt2
          type(parton_t), pointer :: prt, prta, prtb
          real(default) ::  pini(0:3), scale, factor
          integer :: i
          ! print *, " shower_prepare_for_simulate_isr_ana"
          if (.not. associated (prt1%initial) .or. .not. associated (prt2%initial)) then
             return
          end if
          do i = 0, 3
             pini(i) = parton_get_momentum (prt1,i) + parton_get_momentum (prt2,i)
          end do
          scale = - (pini(0)**2 - pini(1)**2 - pini(2)**2 - pini(3)**2)
          call parton_set_simulated (prt1)
          call parton_set_simulated (prt2)
          call shower%add_parent (prt1)
          call shower%add_parent (prt2)
          factor = sqrt (vector4_get_component (prt1%momentum, 0)**2 - scale) / &
                space_part_norm(prt1%momentum)
          prt1%parent%type = prt1%type
          prt1%parent%z = 1._double
          prt1%parent%momentum = prt1%momentum
          prt1%parent%t = scale
          prt1%parent%x = prt1%x
          prt1%parent%initial => prt1%initial
          prt1%parent%belongstoFSR = .false.
          prt1%parent%c1 = prt1%c1
          prt1%parent%c2 = prt1%c2
```

```
prt2%parent%type= prt2%type
prt2%parent%z = 1._double
prt2%parent%momentum = prt2%momentum
prt2%parent%t = scale
prt2%parent%x = prt2%x
prt2%parent%initial => prt2%initial
prt2%parent%belongstoFSR = .false.
prt2%parent%c1 = prt2%c1
prt2%parent%c2 = prt2%c2

do
   call shower_get_first_ISR_scale_for_parton (shower, prt1%parent)
   call shower_get_first_ISR_scale_for_parton (shower, prt2%parent)

   !!! redistribute energy among first partons
   prta => prt1%parent
   prtb => prt2%parent

   do i = 0, 3
      pini(i) = parton_get_momentum (prt1,i) + parton_get_momentum (prt2,i)
   end do
   call parton_set_energy (prta, (pini(0)**2 - prtb%t + prta%t) &
        / (two * pini(0)))
   call parton_set_energy (prtb, pini(0) &
        - vector4_get_component (prta%momentum, 0))

   ! if (abs( (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 0)-&
   !    prt2%parent%x * vector4_get_component(prt1%parent%momentum, 0))/ &
   !    (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 3)-&
   !    prt2%parent%x * vector4_get_component(prt1%parent%momentum, 3)) ).gt. 1._double) then
   !       print *, "cycle", abs( (prt1%parent%x * vector4_get_component(prt2%parent%momentu
   !      prt2%parent%x * vector4_get_component(prt1%parent%momentum, 0))/ &
   !      (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 3)-&
   !      prt2%parent%x * vector4_get_component(prt1%parent%momentum, 3)) )
   !     cycle
   ! else
   exit
   ! end if
end do
!     print *, "BETA+", (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 0)-&
!         prt2%parent%x * vector4_get_component(prt1%parent%momentum, 0))/ &
!         (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 3)-&
!         prt2%parent%x * vector4_get_component(prt1%parent%momentum, 3))
!!! TODO check the code above
call parton_set_simulated (prt1%parent)
call parton_set_simulated (prt2%parent)
!!! rescale momenta
do i = 1, 2
   if (i == 1) then
      prt => prt1%parent
   else
      prt => prt2%parent
   end if
   factor = sqrt (vector4_get_component (prt%momentum,0)**2 - prt%t) &
```

```
            / space_part_norm (prt%momentum)
       prt%momentum = vector4_moving (vector4_get_component (prt%momentum, 0), &
            factor * space_part (prt%momentum))
    end do

    if (prt1%parent%t < 0._double) then
       call shower%add_parent (prt1%parent)
       prt1%parent%parent%momentum = prt1%parent%momentum
       prt1%parent%parent%t = prt1%parent%t
       prt1%parent%parent%x = prt1%parent%x
       prt1%parent%parent%initial => prt1%parent%initial
       prt1%parent%parent%belongstoFSR = .false.
       call shower%add_child (prt1%parent%parent, 2)
    end if

    if (prt2%parent%t < 0._double) then
       call shower%add_parent (prt2%parent)
       prt2%parent%parent%momentum = prt2%parent%momentum
       prt2%parent%parent%t = prt2%parent%t
       prt2%parent%parent%x = prt2%parent%x
       prt2%parent%parent%initial => prt2%parent%initial
       prt2%parent%parent%belongstoFSR = .false.
       call shower%add_child (prt2%parent%parent, 2)
    end if
    ! print *, "after isr_ana_test"
    ! call shower%write ()
    ! print *, "BETA:"
    ! call vector3_write(vector3_canonical(3) * &
    !      ( (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 0)-&
    !       prt2%parent%x * vector4_get_component(prt1%parent%momentum, 0))/ &
    !        (prt1%parent%x * vector4_get_component(prt2%parent%momentum, 3)-&
    !         prt2%parent%x * vector4_get_component(prt1%parent%momentum, 3)) ) )

    ! print *, "after isr_ana_test finished"

  end subroutine shower_prepare_for_simulate_isr_ana_test
```

⟨*Shower core: procedures*⟩+≡
```
  subroutine shower_prepare_for_simulate_isr_ana (shower, prt1, prt2)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout), target :: prt1, prt2
    type(parton_t), pointer :: prt
    real(default) ::  pini(0:3), scale, factor
    real(default) :: oldscales(1:2)
    type(parton_pointer_t) :: temppp
    integer :: i

    if (signal_is_pending ()) return
    ! print *, " shower_prepare_for_simulate_isr_ana"

    if (.not. associated (prt1%initial) .or. &
         .not. associated (prt2%initial)) then
       return
    end if
```

```
do i = 0,3
   pini(i) = parton_get_momentum (prt1,i) + parton_get_momentum (prt2,i)
end do
scale = - (pini(0)**2 - pini(1)**2 - pini(2)**2 - pini(3)**2)

prt1%t = -tscalefactor_isr * abs(scale)
prt2%t = -tscalefactor_isr * abs(scale)
!!! rescale momenta
do i = 1, 2
   if (i == 1) then
      prt => prt1
   else
      prt => prt2
   end if

   factor = sqrt (parton_get_energy (prt)**2 - prt%t) / &
         space_part_norm(prt%momentum)

   prt%momentum = vector4_moving (parton_get_energy (prt), &
         factor * space_part (prt%momentum))
end do

!!! ensure that belongstointeraction bits are set correctly
prt1%belongstointeraction = .true.
prt2%belongstointeraction = .true.

call shower%add_parent (prt1)
call shower%add_parent (prt2)

call parton_set_simulated (prt1)
prt1%parent%type = prt1%type
prt1%parent%z = one
prt1%parent%momentum = prt1%momentum
prt1%parent%t = scale
prt1%parent%x = prt1%x
prt1%parent%initial => prt1%initial
prt1%parent%belongstoFSR = .false.
prt1%parent%c1 = prt1%c1
prt1%parent%c2 = prt1%c2
call shower%add_child (prt1%parent, 2)

call parton_set_simulated (prt2)
prt2%parent%type = prt2%type
prt2%parent%z = one
prt2%parent%momentum = prt2%momentum
prt2%parent%t = scale
prt2%parent%x = prt2%x
prt2%parent%initial => prt2%initial
prt2%parent%belongstoFSR = .false.
prt2%parent%c1 = prt2%c1
prt2%parent%c2 = prt2%c2
call shower%add_child (prt2%parent, 2)
```

```
FIRST_BRANCHINGS: do
   if (signal_is_pending ()) return
   oldscales(1) = prt1%parent%t
   oldscales(2) = prt2%parent%t
   if (abs(prt1%parent%t) > abs(prt2%parent%t)) then
      temppp%p => prt1%parent
   else
      temppp%p => prt2%parent
   end if
   if (.not. parton_is_simulated(temppp%p) .and. .not. parton_is_hadron &
        (temppp%p)) then
      call shower_isr_step (shower, temppp%p)
      if (parton_is_simulated(temppp%p)) then
         ! call parton_generate_ps_ini(prt2%parent)
         if (temppp%p%t < zero) then
            call shower%execute_next_isr_branching (temppp)
            ! call shower%write ()
         else
            call shower_replace_parent_by_hadron (shower, temppp%p%child1)
         end if
      end if
   end if

   if (oldscales(1) == prt1%parent%t .and. &
        oldscales(2) == prt2%parent%t) then
      exit FIRST_BRANCHINGS
   end if
end do FIRST_BRANCHINGS

   ! print *, "  shower_prepare_for_simulate_isr_ana finished"
end subroutine shower_prepare_for_simulate_isr_ana
```

⟨*Shower core: procedures*⟩+≡

```
!! subroutine shower_prepare_for_simulate_fsr_ana (shower, prt1, prt2)
!!    type(shower_t), intent(inout) :: shower
!!    type(parton_t), pointer :: prt1, prt2
!!    real(default) ::  pini(4)
!!    integer i
!!
!!    ! print *, "shower_prepare_for_simulate_fsr_ana"
!!
!!    !!! Define imagined single initiator of shower
!!    call shower%add_child (prt1, 1)
!!    do i = 1, 4
!!       pini(i) = parton_get_momentum (prt1,i - 1) + parton_get_momentum (prt2, i - 1)
!!    end do
!!    call parton_set_simulated (prt1)
!!    call parton_set_child (prt1, prt1%child1, 1)
!!    call parton_set_child (prt1, prt1%child1, 2)
!!    call parton_set_simulated (prt2)
!!    call parton_set_child (prt2, prt1%child1, 1)
!!    call parton_set_child (prt2, prt1%child1, 2)
!!
!!    prt1%child1%type = 94
```

```
!!    prt1%child1%z = parton_get_energy (prt1) / &
!!         (parton_get_energy (prt1) + parton_get_energy (prt2))
!!    call parton_set_simulated (prt1%child1)
!!    call parton_set_parent (prt1%child1, prt1)
!!    call parton_set_momentum (prt1%child1, pini(1), pini(2), pini(3), pini(4))
!!    prt1%child1%t = parton_p4square(prt1%child1)
!!    prt1%child1%costheta = -one
!!
!!    call shower%add_child (prt1%child1, 1)
!!    call shower%add_child (prt1%child1, 2)
!!
!!    prt1%child1%child1%typ = prt1%typ
!!    prt1%child1%child1%momentum = prt1%momentum
!!    prt1%child1%child1%t = prt1%child1%t
!!    call parton_set_parent (prt1%child1%child1, prt1%child1)
!!    prt1%child1%child1%c1 = prt1%c1
!!    prt1%child1%child1%c2 = prt1%c2
!!
!!    prt1%child1%child2%typ = prt2%typ
!!    prt1%child1%child2%momentum = prt2%momentum
!!    prt1%child1%child2%t = prt2%child1%t
!!    call parton_set_parent (prt1%child1%child2, prt1%child1)
!!    prt1%child1%child2%c1 = prt2%c1
!!    prt1%child1%child2%c2 = prt2%c2
!!
!!    ! print *, "  shower_prepare_for_simulate_fsr_ana finished"
!! end subroutine shower_prepare_for_simulate_fsr_ana
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_add_children_of_emitted_timelike_parton (shower, prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), pointer :: prt

  if (prt%t > parton_mass_squared (prt) + D_Min_t) then
     if (parton_is_quark (prt)) then
        !!! q -> qg
        call shower%add_child (prt, 1)
        prt%child1%type = prt%type
        call parton_set_energy (prt%child1, prt%z * parton_get_energy (prt))
        prt%child1%t = prt%t
        call shower%add_child (prt, 2)
        prt%child2%type = 21
        call parton_set_energy (prt%child2, (one - prt%z) * &
             parton_get_energy (prt))
        prt%child2%t = prt%t
     else
        if (int (prt%x) > 0) then
           call shower%add_child (prt, 1)
           prt%child1%type = int (prt%x)
           call parton_set_energy (prt%child1, prt%z * parton_get_energy (prt))
           prt%child1%t = prt%t
           call shower%add_child (prt, 2)
           prt%child2%type = -int (prt%x)
           call parton_set_energy (prt%child2, (one - prt%z) * &
```

```
                parton_get_energy (prt))
            prt%child2%t= prt%t
        else
            call shower%add_child (prt, 1)
            prt%child1%type = 21
            call parton_set_energy (prt%child1, prt%z * parton_get_energy (prt))
            prt%child1%t = prt%t
            call shower%add_child (prt, 2)
            prt%child2%type = 21
            call parton_set_energy (prt%child2, (one - prt%z) * &
                parton_get_energy (prt))
            prt%child2%t = prt%t
        end if
    end if
  end if
end subroutine shower_add_children_of_emitted_timelike_parton
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_simulate_children_ana (shower,prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout) :: prt
  real(default), dimension(1:2) :: t, random, integral
  integer, dimension(1:2) :: gtoqq
  integer :: daughter
  type(parton_t), pointer :: daughterprt
  integer :: n_loop

  if (signal_is_pending ()) return
  gtoqq = 0

  if (D_print) print *, " simulate_children_ana for parton " , prt%nr

  if (.not. associated (prt%child1) .or. .not. associated (prt%child2)) then
      call msg_error ("Shower: error in simulate_children_ana: no children.")
      return
  end if

  ⟨Set beam-remnants and internal partons as simulated⟩

  integral = zero

  !!! impose constraints by angular ordering -> cf. (26) of Gaining analytic control
  !!! check if no branchings are possible
  if (.not. prt%child1%simulated) then
      prt%child1%t = min (prt%child1%t, &
          0.5_default * parton_get_energy (prt%child1)**2 * (one - &
          parton_get_costheta (prt)))
      if (min (prt%child1%t, parton_get_energy (prt%child1)**2) < &
          parton_mass_squared (prt%child1) + D_Min_t) then
        prt%child1%t = parton_mass_squared (prt%child1)
        call parton_set_simulated (prt%child1)
      end if
  end if
  if (.not. prt%child2%simulated) then
```

```
      prt%child2%t = min (prt%child2%t, &
            0.5_default * parton_get_energy (prt%child2)**2 * (one - &
            parton_get_costheta (prt)))
      if (min (prt%child2%t, parton_get_energy (prt%child2)**2) < &
            parton_mass_squared (prt%child2) + D_Min_t) then
         prt%child2%t = parton_mass_squared (prt%child2)
         call parton_set_simulated (prt%child2)
      end if
   end if
end if

call tao_random_number (random)

n_loop = 0
do
   if (signal_is_pending ()) return
   n_loop = n_loop + 1
   if (n_loop > 900) then
      !!! try with massless quarks
      treat_duscb_quarks_massless = .true.
   end if
   if (n_loop > 1000) then
      call shower%write ()
      call msg_message (" simulate_children_ana failed for parton ", prt%nr)
      call msg_error ("BUG: too many loops in simulate_children_ana (?)")
      shower%valid = .false.
      return
   end if

   t(1) = prt%child1%t
   t(2) = prt%child2%t

   !!! check if a branching in the range t(i) to t(i) - tstep(i) occurs

   !!! check for child1
   if (.not. parton_is_simulated (prt%child1)) then
      call parton_simulate_stept &
            (prt%child1, integral(1), random(1), gtoqq(1))
   end if
   !!! check for child2
   if (.not. parton_is_simulated(prt%child2)) then
      call parton_simulate_stept &
            (prt%child2, integral(2), random(2), gtoqq(2))
   end if

   if (parton_is_simulated (prt%child1) .and. &
       parton_is_simulated (prt%child2)) then
      if (sqrt (prt%t) <= sqrt (prt%child1%t) + sqrt (prt%child2%t)) then
         ⟨Repeat the simulation for the parton with the lower virtuality⟩
      else
         exit
      end if
   end if
end do
```

2218

```
      call parton_apply_costheta (prt)

      ⟨Add children⟩
      call shower_parton_update_color_connections (shower, prt)
    end subroutine shower_simulate_children_ana
```

⟨*Set beam-remnants and internal partons as simulated*⟩≡
```
  if (abs (prt%type) >= 90 .and. abs (prt%type) <= 93) then
     !!! prt is beam-remnant
     call parton_set_simulated (prt)
     return
  end if

  !!! check if partons are "internal" -> fixed scale
  if (prt%child1%type == 94) then
     call parton_set_simulated (prt%child1)
  end if
  if (prt%child2%type == 94) then
     call parton_set_simulated (prt%child2)
  end if
```

⟨*Repeat the simulation for the parton with the lower virtuality*⟩≡
```
  !!! virtuality : t - m**2 (assuming it's not fixed)
  if (prt%child1%type == 94 .and. prt%child2%type == 94) then
     call msg_fatal &
          ("Shower: both partons fixed, but momentum not conserved")
  else if (prt%child1%type == 94) then
     !!! reset child2
     call parton_set_simulated (prt%child2, .false.)
     prt%child2%t = min (prt%child1%t, (sqrt (prt%t) - &
          sqrt (prt%child1%t))**2)
     integral(2) = zero
     call tao_random_number (random(2))
  else if (prt%child2%type.eq.94) then
     ! reset child1
     call parton_set_simulated (prt%child1, .false.)
     prt%child1%t = min (prt%child2%t, (sqrt (prt%t) - &
          sqrt (prt%child2%t))**2)
     integral(1) = zero
     call tao_random_number (random(1))
  else if (prt%child1%t - parton_mass_squared (prt%child1) > &
       prt%child2%t - parton_mass_squared (prt%child2)) then
     !!! reset child2
     call parton_set_simulated (prt%child2, .false.)
     prt%child2%t = min (prt%child1%t, (sqrt (prt%t) - &
          sqrt (prt%child1%t))**2)
     integral(2) = zero
     call tao_random_number (random(2))
  else
     !!! reset child1 ! TODO choose child according to their t
     call parton_set_simulated (prt%child1, .false.)
     prt%child1%t = min (prt%child2%t, (sqrt (prt%t) - &
          sqrt (prt%child2%t))**2)
     integral(1) = zero
```

```
            call tao_random_number (random(1))
         end if

⟨Add children⟩≡
   do daughter = 1, 2
      if (signal_is_pending ()) return
      if (daughter == 1) then
         daughterprt => prt%child1
      else
         daughterprt => prt%child2
      end if
      if (daughterprt%t < parton_mass_squared (daughterprt) + D_Min_t) then
         cycle
      end if
      if (.not. (parton_is_quark (daughterprt) .or. &
           parton_is_gluon (daughterprt))) then
         cycle
      end if
      if (parton_is_quark (daughterprt)) then
         !!! q -> qg
         call shower%add_child (daughterprt, 1)
         daughterprt%child1%type = daughterprt%type
         call parton_set_energy (daughterprt%child1, &
              daughterprt%z * parton_get_energy (daughterprt))
         daughterprt%child1%t = daughterprt%t
         call shower%add_child (daughterprt, 2)
         daughterprt%child2%type = 21
         call parton_set_energy (daughterprt%child2, (one - &
              daughterprt%z) * parton_get_energy (daughterprt))
         daughterprt%child2%t = daughterprt%t
      else if (parton_is_gluon (daughterprt)) then
         if (gtoqq(daughter) > 0) then
            call shower%add_child (daughterprt, 1)
            daughterprt%child1%type = gtoqq (daughter)
            call parton_set_energy (daughterprt%child1, &
                 daughterprt%z * parton_get_energy (daughterprt))
            daughterprt%child1%t = daughterprt%t
            call shower%add_child (daughterprt, 2)
            daughterprt%child2%type = - gtoqq (daughter)
            call parton_set_energy (daughterprt%child2, (one - &
                 daughterprt%z) * parton_get_energy (daughterprt))
            daughterprt%child2%t = daughterprt%t
         else
            call shower%add_child (daughterprt, 1)
            daughterprt%child1%type = 21
            call parton_set_energy (daughterprt%child1, &
                 daughterprt%z * parton_get_energy (daughterprt))
            daughterprt%child1%t = daughterprt%t
            call shower%add_child (daughterprt, 2)
            daughterprt%child2%type = 21
            call parton_set_energy (daughterprt%child2, (one - &
                 daughterprt%z) * parton_get_energy (daughterprt))
            daughterprt%child2%t = daughterprt%t
         end if
      end if
```

```
      end do
```

⟨*Shower core: procedures*⟩+≡

```
    subroutine shower_generate_next_fsr_branchings (shower)
      type(shower_t), intent(inout) :: shower
      integer i, index
      type(parton_t),  pointer :: prt

      !!! find mother with highest t to be simulated
      index = 0
      do i = 1, size (shower%partons)
         prt => shower%partons(i)%p
         if (.not. prt%belongstoFSR) cycle
         if (prt%belongstointeraction) cycle
         if (associated(prt%child1) .and. associated(prt%child2)) then
            if (parton_is_simulated (prt%child1) .and. &
                parton_is_simulated (prt%child2)) cycle
         end if
         if (parton_is_final (prt)) cycle

            index = i
            exit
      end do

      if (index.eq.0) then
         call msg_message ("Shower: no branchable partons found")
         return
      end if

      prt => shower%partons(index)%p
      call shower_simulate_children_ana(shower, prt)

    end subroutine shower_generate_next_fsr_branchings
```

The recoiler is `otherprt`. Instead of the random number and the exponential of the integral, we compare the logarithm of the random number and the integral.

⟨*Shower core: procedures*⟩+≡

```
    subroutine shower_isr_step_pt (shower, prt)
      type(shower_t), intent(inout) :: shower
      type(parton_t), target, intent(inout) :: prt
      type(parton_t), pointer :: otherprt

      real(default) :: scale, scalestep
      real(default) :: integral, random, factor
      real(default) :: temprand1, temprand2

      otherprt => shower_find_recoiler (shower, prt)

      scale = prt%scale
      call tao_random_number (temprand1)
      call tao_random_number (temprand2)
      scalestep = max (abs (scalefactor1 * scale) * temprand1, &
            scalefactor2 * temprand2 * D_Min_scale)
      call tao_random_number (random)
```

2221

```
    random = - twopi * log(random)
    integral = zero

    if (scale - 0.5_default * scalestep < D_Min_scale) then
       !!! close enough to cut-off scale -> ignore
       prt%scale = zero
       prt%t = parton_mass_squared (prt)
       call parton_set_simulated (prt)
    else
       prt%scale = scale - 0.5_default * scalestep
       factor = scalestep * (D_alpha_s_isr (prt%scale) / (prt%scale * &
            get_pdf (prt%initial%type, prt%x, prt%scale, prt%type)))
       integral = integral + factor * integral_over_z_isr_pt &
            (prt, otherprt, (random - integral) / factor)
       if (integral > random) then
          !!! prt%scale set above and prt%z set in integral_over_z_isr_pt
          call parton_set_simulated (prt)
          prt%t = - prt%scale / (one - prt%z)
       else
          prt%scale = scale - scalestep
       end if
    end if

contains

  function integral_over_z_isr_pt (prt, otherprt, final) &
       result (integral)
    type(parton_t), intent(inout) :: prt, otherprt
    real(default), intent(in) :: final
    real(default) :: integral
    real(default) :: mbr, r
    real(default) :: zmin, zmax, z, zstep
    integer :: n_bin
    integer, parameter :: n_total_bins = 100
    real(default) :: quarkpdfsum
    real(default) :: temprand
    integer :: quark

    quarkpdfsum = zero
    if (D_print) then
       print *, "integral_over_z_isr_pt for scale = ", prt%scale
    end if

    integral = zero
    mbr = (prt%momentum + otherprt%momentum)**1
    zmin = prt%x
    zmax = min (one - (sqrt (prt%scale) / mbr) * &
         (sqrt(one + 0.25_default * prt%scale / mbr**2) - &
         0.25_default * sqrt(prt%scale) / mbr), maxz_isr)
    zstep = (zmax - zmin) / n_total_bins

    if (zmin > zmax) then
       !!! print *, " error in integral_over_z_isr_pt: zmin > zmax ", &
       !!!          zmin, zmax, prt%scale, mbr
```

```
         integral = zero
         return
      end if

!!! divide the range [zmin:zmax] in n_total_bins ->
BINS: do n_bin = 1, n_total_bins
   z = zmin + zstep * (n_bin -0.5_default)
   !!! z-value in the middle of the bin

   if (parton_is_gluon (prt)) then
      QUARKS: do quark = -D_Nf, D_Nf
         if (quark == 0) cycle quarks
         quarkpdfsum = quarkpdfsum + get_pdf &
             (prt%initial%type, prt%x / z, prt%scale, quark)
      end do QUARKS
      !!! g -> gg or q -> gq
      integral = integral + (zstep / z) * ((P_ggg (z) + &
          P_ggg (one - z)) * get_pdf (prt%initial%type, &
          prt%x / z, prt%scale, 21) + P_qqg (one - z) * quarkpdfsum)
   else if (parton_is_quark (prt)) then
      !!! q -> qg or g -> qq
      integral = integral + (zstep / z) * ( P_qqg (z) * &
          get_pdf (prt%initial%type, prt%x / z, prt%scale, prt%type) + &
          P_gqq(z) * get_pdf (prt%initial%type, prt%x / z, prt%scale, 21))
   else
      ! call msg_fatal ("Bug neither quark nor gluon in" &
      !            // " integral_over_z_isr_pt")
   end if
   if (integral > final) then
      prt%z = z
      call tao_random_number (temprand)
      !!! decide typ of father partons
        if (parton_is_gluon (prt)) then
           if (temprand > (P_qqg (one - z) * quarkpdfsum) / &
               ((P_ggg (z) + P_ggg (one - z)) * get_pdf &
               (prt%initial%type, prt%x / z, prt%scale, 21) &
               + P_qqg (one - z) * quarkpdfsum)) then
              !!! gluon => gluon + gluon
             prt%aux_pt = 21
           else
              !!! quark => quark + gluon
              !!! decide which quark flavor the parent is
             r = temprand * quarkpdfsum
             WHICH_QUARK: do quark = -D_Nf, D_Nf
                if (quark == 0) cycle WHICH_QUARK
                if (r > quarkpdfsum - get_pdf (prt%initial%type, &
                    prt%x / z, prt%scale, quark)) then
                   prt%aux_pt = quark
                   exit WHICH_QUARK
                else
                   quarkpdfsum = quarkpdfsum - get_pdf &
                       (prt%initial%type, prt%x / z, prt%scale, quark)
                end if
             end do WHICH_QUARK
```

```
                         end if

              else if (parton_is_quark (prt)) then
                  if (temprand > (P_qqg (z) * get_pdf (prt%initial%type, &
                       prt%x / z, prt%scale, prt%type)) / &
                       (P_qqg (z) * get_pdf (prt%initial%type, prt%x / z, &
                       prt%scale, prt%type) + &
                       P_gqq (z) * get_pdf (prt%initial%type, prt%x / z, &
                       prt%scale, 21))) then
                     !!! gluon => quark + antiquark
                     prt%aux_pt = 21
                  else
                     !!! quark => quark + gluon
                     prt%aux_pt = prt%type
                  end if
              end if
              exit BINS
          end if
      end do BINS
    end function integral_over_z_isr_pt
  end subroutine shower_isr_step_pt
```

This function returns a pointer to the parton with the next ISR branching, while
FSR branchings are ignored.

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: generate_next_isr_branching_veto => &
        shower_generate_next_isr_branching_veto
```

⟨*Shower core: procedures*⟩+≡
```
    function shower_generate_next_isr_branching_veto &
         (shower) result (next_brancher)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t) :: next_brancher
      integer :: i
      type(parton_t),  pointer :: prt
      real(default) :: random
      !!! pointers to branchable partons
      type(parton_pointer_t), dimension(:), allocatable :: partons
      integer :: n_partons
      real(default) :: weight
      real(default) :: temp1, temp2, temp3, E3

      if (signal_is_pending ()) return

      if (isr_pt_ordered) then
         next_brancher = shower%generate_next_isr_branching ()
         return
      end if
      next_brancher%p => null()
      !!! branchable partons
      n_partons = 0
      do i = 1,size (shower%partons)
         prt => shower%partons(i)%p
         if (signal_is_pending ()) return
```

```
      if (.not. associated (prt)) cycle
      if (prt%belongstoFSR) cycle
      if (parton_is_final (prt)) cycle
      if (.not. prt%belongstoFSR .and. parton_is_simulated (prt)) cycle
      n_partons = n_partons + 1
end do
!!! print *, "n_partons = ", n_partons
if (n_partons == 0) then
   return
end if
allocate (partons(1:n_partons))
n_partons = 1
do i = 1, size (shower%partons)
   prt => shower%partons(i)%p
   if (.not. associated (prt)) cycle
   if (prt%belongstoFSR) cycle
   if (parton_is_final (prt)) cycle
   if (.not. prt%belongstoFSR .and. parton_is_simulated (prt)) cycle
   if (signal_is_pending ()) return
   partons(n_partons)%p => shower%partons(i)%p
   n_partons = n_partons + 1
end do
! print *, "trials"
!!! generate initial trial scales
do i = 1, size (partons)
   if (signal_is_pending ()) return
   call generate_next_trial_scale (partons(i)%p)
end do
! print *, "trials finished"

do
   ! call shower%write ()
   !!! search for parton with the highest trial scale
   prt => partons(1)%p
   do i = 1, size (partons)
      if (prt%t >= zero) cycle
      if (abs (partons(i)%p%t) > abs (prt%t)) then
         prt => partons(i)%p
      end if
   end do

   if (prt%t >= zero) then
      next_brancher%p => null()
      exit
   end if
   !!! generate trial z and type of mother prt
   call generate_trial_z_and_typ (prt)

   !!! weight with pdf and alpha_s

   temp1 = (D_alpha_s_isr ((one - prt%z) * abs(prt%t)) &
        / sqrt (alphasxpdfmax))
   temp2 = get_xpdf (prt%initial%type, prt%x, prt%t, &
        prt%type) / sqrt (alphasxpdfmax)
```

```fortran
      temp3 = get_xpdf (prt%initial%type, prt%child1%x, prt%child1%t, &
          prt%child1%type) / &
          get_xpdf (prt%initial%type, prt%child1%x, prt%t, prt%child1%type)
      if (temp1 * temp2 * temp3 > one) then
         print *, "weights:", temp1, temp2, temp3
      end if
      weight = (D_alpha_s_isr ((one - prt%z) * abs(prt%t))) * &
          get_xpdf (prt%initial%type, prt%x, prt%t, prt%type) * &
          get_xpdf (prt%initial%type, prt%child1%x, prt%child1%t, &
          prt%child1%type) / &
          get_xpdf (prt%initial%type, prt%child1%x, prt%t, prt%child1%type)
      if (weight > alphasxpdfmax) then
         print *, "Setting alphasxpdfmax from ", alphasxpdfmax, " to ", weight
         alphasxpdfmax = weight
      end if
      weight = weight / alphasxpdfmax
      call tao_random_number (random)
      ! print *, weight , ">", random, "?"
      if (weight < random) then
         !!! discard branching
         call generate_next_trial_scale (prt)
         cycle
      end if
      !!! branching accepted so far
      !!! generate emitted parton
      prt%child2%t = abs(prt%t)
      call parton_set_energy (prt%child2, sqrt (abs(prt%t)))
      if (isr_only_onshell_emitted_partons) then
         prt%child2%t = parton_mass_squared(prt%child2)
      else
         call parton_next_t_ana (prt%child2)
      end if

      if (thetabar (prt, shower_find_recoiler (shower, prt), E3)) then
         !!! setting energies
         call parton_set_energy (prt, E3)
         call parton_set_energy (prt%child2, &
             E3 - parton_get_energy (prt%child1))

         !!! found branching
         call parton_generate_ps_ini (prt)
         next_brancher%p => prt
         call parton_set_simulated (prt)
         exit
      else
         call generate_next_trial_scale (prt)
         cycle
      end if
   end do
if (.not. associated (next_brancher%p)) then
   !!! no further branching found -> all partons emitted by hadron
   print *, "--all partons emitted by hadrons---"
   do i = 1, size(partons)
      call shower_replace_parent_by_hadron (shower, partons(i)%p%child1)
```

```
      end do
   end if
   !!! some bookkeeping
   call shower%sort_partons ()
   ! call shower_boost_to_CMframe(shower)          ! really necessary?
   ! call shower_rotate_to_z(shower)               ! really necessary?
contains

   subroutine generate_next_trial_scale(prt)
     type(parton_t), pointer, intent(inout) :: prt
     real(default) :: random, F
     real(default) :: zmax = 0.99_default !! ??
     ! print *, "generate next_trial"
     call tao_random_number (random)
     F = one    !!! TODO
     F = alphasxpdfmax / (two * pi)
     if (parton_is_quark (prt%child1)) then
        F = F * (integral_over_P_gqq (prt%child1%x, zmax) + &
                 integral_over_P_qqg (prt%child1%x, zmax))
     else if (parton_is_gluon (prt%child1)) then
        F = F * (integral_over_P_ggg (prt%child1%x, zmax) + &
             two * D_Nf * &
             integral_over_P_qqg (one - zmax, one - prt%child1%x))
     else
        print *, "neither quark nor gluon in generate_next_trial_scale"
     end if
     F = F / get_xpdf (prt%child1%initial%type, prt%child1%x, &
         prt%child1%t, prt%child1%type)
     prt%t = prt%t * random**(one / F)
     if (abs (prt%t) - parton_mass_squared (prt) < D_Min_t) then
        prt%t = parton_mass_squared (prt)
     end if
     ! print *, "generate next_trial finished"
   end subroutine generate_next_trial_scale

   subroutine generate_trial_z_and_typ(prt)
     type(parton_t), pointer, intent(inout) :: prt
     real(default) :: random
     real(default) :: z, zstep, zmin, integral
     real(default) :: zmax = 0.99_default !! ??
     ! print *, "generate next_z"
     call tao_random_number(random)
     integral = zero
     !!! decide which branching a->bc occurs
     if (parton_is_quark (prt%child1)) then
        if (random < integral_over_P_qqg (prt%child1%x, zmax) / &
             (integral_over_P_qqg (prt%child1%x, zmax) + &
              integral_over_P_gqq(prt%child1%x, zmax))) then
           prt%type = prt%child1%type
           prt%child2%type = 21
           integral = integral_over_P_qqg (prt%child1%x, zmax)
        else
           prt%type = 21
           prt%child2%type = - prt%child1%type
```

```
         integral = integral_over_P_gqq (prt%child1%x, zmax)
      end if
else if (parton_is_gluon (prt%child1)) then
   if (random < integral_over_P_ggg (prt%child1%x, zmax) / &
        (integral_over_P_ggg (prt%child1%x, zmax) + two * D_Nf * &
         integral_over_P_qqg (one - zmax, &
         one - prt%child1%x))) then
      prt%type = 21
      prt%child2%type = 21
      integral = integral_over_P_ggg (prt%child1%x, zmax)
   else
      call tao_random_number (random)
      prt%type = 1 + floor(random * D_Nf)
      call tao_random_number (random)
      if (random > 0.5_default) prt%type = - prt%type
      prt%child2%type = prt%type
      integral = integral_over_P_qqg (one - zmax, &
            one - prt%child1%x)
   end if
else
   print *, "neither quark nor gluon in generate_next_trial_scale"
end if
!!! generate the z-value
!!! z between prt%child1%x and zmax
! prt%z = one - random * (one - prt%child1%x)        ! TODO

call tao_random_number(random)
zmin = prt%child1%x
zstep = max(0.1_default, 0.5_default * (zmax - zmin))
z = zmin
! print *, zmin, z
if (zmin > zmax) then
   print *, " zmin = ", zmin, " zmax = ", zmax
   call msg_fatal ("Shower: zmin greater than zmax")
end if
!!! procedure pointers would be helpful here
if (parton_is_quark (prt) .and. parton_is_quark (prt%child1)) then
   do
      ! print *, 1, z, zstep, zmin, zmax
      zstep = min(zstep, 0.5_default * (zmax - z))
      if (abs(zstep) < 0.00001) exit
      if (integral_over_P_qqg (zmin, z) < random * integral) then
         if (integral_over_P_qqg (zmin, min(z + zstep, zmax)) &
              < random * integral) then
            z = min (z + zstep, zmax)
            cycle
         else
            zstep = zstep * 0.5_default
            cycle
         end if
      end if
   end do
else if (parton_is_quark (prt) .and. parton_is_gluon (prt%child1)) then
   do
```

```
            ! print *, 2, z, zstep
            zstep = min(zstep, 0.5_default * (zmax - z))
            if (abs(zstep) < 0.00001) exit
            if (integral_over_P_qqg (zmin, z) < random * integral) then
               if (integral_over_P_qqg (zmin, min(z + zstep, zmax)) &
                    < random * integral) then
                  z = min(z + zstep, zmax)
                  cycle
               else
                  zstep = zstep * 0.5_default
                  cycle
               end if
            end if
         end do
      else if (parton_is_gluon (prt) .and. parton_is_quark (prt%child1)) then
         do
            ! print *, 3, z, zstep
            zstep = min(zstep, 0.5_default * (zmax - z))
            if (abs (zstep) < 0.00001) exit
            if (integral_over_P_gqq (zmin, z) < random * integral) then
               if (integral_over_P_gqq (zmin, min(z + zstep, zmax)) &
                    < random * integral) then
                  z = min (z + zstep, zmax)
                  cycle
               else
                  zstep = zstep * 0.5_default
                  cycle
               end if
            end if
         end do
      else if (parton_is_gluon (prt) .and. parton_is_gluon (prt%child1)) then
         do
            ! print *, 4, z, zstep
            zstep = min(zstep, 0.5_default * (zmax - z))
            if (abs (zstep) < 0.00001) exit
            if (integral_over_P_ggg(zmin, z) < random * integral) then
               if (integral_over_P_ggg (zmin, min(z + zstep, zmax)) &
                    < random * integral) then
                  z = min(z + zstep, zmax)
                  cycle
               else
                  zstep = zstep * 0.5_default
                  cycle
               end if
            end if
         end do
      else
      end if
      prt%z = z
      prt%x = prt%child1%x / prt%z
      ! print *, "generate next_z finished"
    end subroutine generate_trial_z_and_typ
  end function shower_generate_next_isr_branching_veto
```

⟨Shower core: procedures⟩+≡

```
function shower_find_recoiler (shower, prt) result(recoiler)
  type(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout), target :: prt
  type(parton_t), pointer :: recoiler
  type(parton_t), pointer :: otherprt1, otherprt2
  integer :: n_int
  logical :: goon
  otherprt1 => null()
  otherprt2 => null()
  DO_INTERACTIONS: do n_int = 1, size(shower%interactions)
     otherprt1=>shower%interactions(n_int)%i%partons(1)%p
     otherprt2=>shower%interactions(n_int)%i%partons(2)%p
     PARTON1: do
        if (associated (otherprt1%parent)) then
           if (.not. parton_is_hadron(otherprt1%parent) .and. &
                parton_is_simulated (otherprt1%parent)) then
              otherprt1 => otherprt1%parent
              if (associated (otherprt1, prt)) then
                 exit PARTON1
              end if
           else
              exit PARTON1
           end if
        else
           exit PARTON1
        end if
     end do PARTON1
     PARTON2: do
        if (associated (otherprt2%parent)) then
           if (.not. parton_is_hadron(otherprt2%parent) .and. &
                parton_is_simulated (otherprt2%parent)) then
              otherprt2 => otherprt2%parent
              if (associated (otherprt2, prt)) then
                 exit PARTON2
              end if
           else
              exit PARTON2
           end if
        else
           exit PARTON2
        end if
     end do PARTON2

     if (associated (otherprt1, prt) .or. associated (otherprt2, prt)) then
        exit DO_INTERACTIONS
     end if
     if (associated (otherprt1%parent, prt) .or. &
          associated (otherprt2%parent, prt)) then
        exit DO_INTERACTIONS
     end if
  end do DO_INTERACTIONS

  recoiler => null()
```

2230

```
         if (associated (otherprt1%parent, prt)) then
            recoiler => otherprt2
         else if (associated (otherprt2%parent, prt)) then
            recoiler => otherprt1
         else if (associated (otherprt1, prt)) then
            recoiler => otherprt2
         else if (associated (otherprt2, prt)) then
            recoiler => otherprt1
         else
            call shower%write ()
            call parton_write (prt)
            call msg_error ("Shower: no otherparton found")
         end if
      end function shower_find_recoiler
```

⟨*Shower core: procedures*⟩+≡

```
      subroutine shower_isr_step (shower, prt)
        type(shower_t), intent(inout) :: shower
        type(parton_t), target, intent(inout) :: prt
        type(parton_t), pointer :: otherprt => null()
        real(default) :: t, tstep
        real(default) :: integral, random
        real(default) :: temprand1, temprand2
        ! print *, "shower_isr_step for parton ", prt%nr
        otherprt => shower_find_recoiler(shower, prt)
        ! if (.not. otherprt%child1%belongstointeraction) then
        !    otherprt => otherprt%child1
        ! end if

        if (signal_is_pending ()) return
        t = max(prt%t, prt%child1%t)
        call tao_random_number(random)
        ! compare Integral and log(random) instead of random and exp(-Integral)
        random = - twopi * log(random)
        integral = zero
        call tao_random_number (temprand1)
        call tao_random_number (temprand2)
        tstep = max (abs (0.02_default * t) * temprand1, &
             0.02_default * temprand2 * D_Min_t)
        if (t + 0.5_default * tstep > - D_Min_t) then
           prt%t = parton_mass_squared (prt)
           call parton_set_simulated (prt)
        else
           prt%t = t + 0.5_default * tstep
           integral = integral + tstep * &
                integral_over_z_isr (prt, otherprt,(random - integral) / tstep)
           if (integral > random) then
              prt%t = t + 0.5_default * tstep
              prt%x = prt%child1%x / prt%z
              call parton_set_simulated (prt)
           else
              prt%t = t + tstep
           end if
        end if
```

```fortran
    ! print *, "  shower_isr_step finished"

contains
  function integral_over_z_isr (prt, otherprt, final) result (integral)
    type(parton_t), intent(inout) :: prt, otherprt
    real(default), intent(in) :: final
    real(default) integral
    real(default) :: minz, maxz, z, shat,s
    integer :: quark

    !!! calculate shat -> s of parton-parton system
    shat= (otherprt%momentum + prt%child1%momentum)**2
    !!! calculate s -> s of hadron-hadron system
    s= (otherprt%initial%momentum + prt%initial%momentum)**2
    integral = zero
    minz = prt%child1%x
    maxz = maxzz (shat, s)

    !!! for gluon
    if (parton_is_gluon (prt%child1)) then
       !!! 1: g->gg
       prt%type = 21
       prt%child2%type = 21
         z = minz
         prt%child2%t = abs(prt%t)
         call integral_over_z_part_isr &
              (prt,otherprt, shat, minz, maxz, integral, final)
         if (integral > final) then
            ! print *, "prt%type = ", prt%type
            return
         end if
         !!! 2: q->gq
         do quark = -D_Nf, D_Nf
            if (quark == 0) cycle
            prt%type = quark
            prt%child2%type = quark
            z = minz
            prt%child2%t = abs(prt%t)
            call integral_over_z_part_isr &
                 (prt,otherprt, shat, minz, maxz, integral, final)
            if (integral > final) then
               ! print *, "prt%type = ", prt%type
               return
            end if
         end do
      else if (parton_is_quark (prt%child1)) then
         !!! 1: q->qg
         prt%type = prt%child1%type
         prt%child2%type = 21
         z = minz
         prt%child2%t = abs(prt%t)
         call integral_over_z_part_isr &
              (prt,otherprt, shat, minz, maxz, integral, final)
```

2232

```
      if (integral > final) then
         ! print *, "prt%type = ", prt%type
         return
      end if
      !!! 2: g->qqbar
      prt%type = 21
      prt%child2%type = -prt%child1%type
      z = minz
      prt%child2%t = abs(prt%t)
      call integral_over_z_part_isr &
           (prt,otherprt, shat, minz, maxz, integral, final)
      if (integral > final) then
         ! print *, "prt%type = ", prt%type
         return
      end if
   end if
end function integral_over_z_isr

subroutine integral_over_z_part_isr &
     (prt, otherprt, shat ,minz, maxz, retvalue, final)
  type(parton_t), intent(inout) :: prt, otherprt
  real(default), intent(in) :: shat, minz, maxz, final
  real(default), intent(inout) :: retvalue
  real(default) :: z, zstep
  real(default) :: r1,r3,s1,s3
  real(default) :: pdf_divisor
  real(default) :: temprand
  real(default), parameter :: zstepfactor = 0.1_default
  real(default), parameter :: zstepmin = 0.0001_default
  if (D_print) print *, "integral_over_z_part_isr"
  pdf_divisor = get_pdf &
       (prt%initial%type, prt%child1%x, prt%t, prt%child1%type)
  z = minz
  s1 = shat + abs(otherprt%t) + abs(prt%child1%t)
  r1 = sqrt (s1**2 - four * abs(otherprt%t * prt%child1%t))
  ZLOOP: do
     if (signal_is_pending ()) return
     if (z >= maxz) then
        exit
     end if
     call tao_random_number (temprand)
     if (parton_is_gluon (prt%child1)) then
        if (parton_is_gluon (prt)) then
           !!! g-> gg -> divergencies at z->0 and z->1
           zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
        else
           !!! q-> gq -> divergencies at z->0
           zstep = max(zstepmin, temprand * zstepfactor * (one - z))
        end if
     else
        if (parton_is_gluon (prt)) then
           !!! g-> qqbar -> no divergencies
           zstep = max(zstepmin, temprand * zstepfactor)
        else
```

```
                     !!! q-> qg -> divergencies at z->1
                     zstep = max(zstepmin, temprand * zstepfactor * (one - z))
                 end if
             end if
             zstep = min(zstep, maxz - z)
             prt%z = z + 0.5_default * zstep
             s3 = shat / prt%z + abs(otherprt%t) + abs(prt%t)
             r3 = sqrt (s3**2 - four * abs(otherprt%t * prt%t))
             !!! TODO: WHY is this if needed?
             if (otherprt%t /= zero) then
                 prt%child2%t = min ((s1 * s3 - r1 * r3) / &
                         (two * abs(otherprt%t)) - abs(prt%child1%t) - &
                         abs(prt%t), abs(prt%child1%t))
             else
                 prt%child2%t = abs(prt%child1%t)
             end if
             do
                 call parton_set_energy (prt%child2, sqrt (abs(prt%child2%t)))
                 if (isr_only_onshell_emitted_partons) then
                     prt%child2%t = parton_mass_squared (prt%child2)
                 else
                     call parton_next_t_ana (prt%child2)
                 end if
                 !!! take limits by recoiler into account
                 call parton_set_energy (prt, (shat / prt%z + &
                         abs(otherprt%t) - abs(prt%child1%t) - &
                         prt%child2%t) / (two * sqrt(shat)))
                 call parton_set_energy (prt%child2, &
                         parton_get_energy (prt) - parton_get_energy (prt%child1))
                 !!! check if E and t of prt%child2 are consistent
                 if (parton_get_energy (prt%child2)**2 < prt%child2%t &
                         .and. prt%child2%t > parton_mass_squared (prt%child2)) then
                     !!! E is too small to have p_T^2 = E^2 - t > 0
                     !!!         -> cycle to find another solution
                     cycle
                 else
                     !!! E is big enough -> exit
                     exit
                 end if
             end do
             if (thetabar (prt, otherprt) .and. pdf_divisor > zero &
                     .and. parton_get_energy (prt%child2) > zero) then
                 retvalue = retvalue + (zstep / prt%z) * &
                         (D_alpha_s_isr ((one - prt%z) * prt%t) * &
                         P_prt_to_child1 (prt) * &
                         get_pdf (prt%initial%type, prt%child1%x / prt%z, &
                         prt%t, prt%type)) / (abs(prt%t) * pdf_divisor)
             end if
             if (retvalue > final) then
                 exit
             else
                 z = z + zstep
             end if
         end do ZLOOP
```

2234

```
              end subroutine integral_over_z_part_isr
           end subroutine shower_isr_step
```

This returns a pointer to the parton with the next ISR branching, again FSR
branchings are ignored.

⟨*Shower core: shower: TBP*⟩+≡
```
      procedure :: generate_next_isr_branching => &
          shower_generate_next_isr_branching
```

⟨*Shower core: procedures*⟩+≡
```
      function shower_generate_next_isr_branching &
          (shower) result (next_brancher)
        class(shower_t), intent(inout) :: shower
        type(parton_pointer_t) :: next_brancher
        integer i, index
        type(parton_t),  pointer :: prt
        real(default) :: maxscale
        next_brancher%p => null()
        do
           if (signal_is_pending ()) return
           if (shower_isr_is_finished (shower)) exit
           !!! find mother with highest |t| or pt to be simulated
           index = 0
           maxscale = zero
           call shower%sort_partons ()
           do i = 1,size (shower%partons)
              prt => shower%partons(i)%p
              if (.not. associated (prt)) cycle
              if (.not. isr_pt_ordered) then
                 if (prt%belongstointeraction) cycle
              end if
              if (prt%belongstoFSR) cycle
              if (parton_is_final (prt)) cycle
              if (.not. prt%belongstoFSR .and. parton_is_simulated (prt)) cycle
              index = i
              exit
           end do
           if (index == 0) then
              ! print *, " no branchable partons found"
              return
           end if

           prt => shower%partons(index)%p

           !!! ISR simulation
           if (isr_pt_ordered) then
              call shower_isr_step_pt (shower, prt)
           else
              call shower_isr_step (shower, prt)
           end if
           if (parton_is_simulated (prt)) then
              if (prt%t < zero) then
                 next_brancher%p => prt
                 if (.not. isr_pt_ordered) call parton_generate_ps_ini (prt)
```

```
                    exit
              else
                  if (.not. isr_pt_ordered) then
                      call shower_replace_parent_by_hadron (shower, prt%child1)
                  else
                      call shower_replace_parent_by_hadron (shower, prt)
                  end if
              end if
          end if
      end do

      !!! some bookkeeping
      call shower%sort_partons ()
      call shower_boost_to_CMframe (shower)           !!! really necessary?
      call shower_rotate_to_z (shower)                !!! really necessary?
      ! print *, "shower_generate_next_isr_branching finished"
      end function shower_generate_next_isr_branching
```

This is a loop which searches for all emitted and branched partons.

⟨*Shower core: shower: TBP*⟩+≡
```
      procedure :: generate_fsr_for_isr_partons => &
          shower_generate_fsr_for_partons_emitted_in_ISR
```

⟨*Shower core: procedures*⟩+≡
```
      subroutine shower_generate_fsr_for_partons_emitted_in_ISR (shower)
        class(shower_t), intent(inout) :: shower
        integer :: n_int, i
        type(parton_t), pointer :: prt
        if (isr_only_onshell_emitted_partons) return

        INTERACTIONS_LOOP: do n_int = 1, size (shower%interactions)
           INCOMING_PARTONS_LOOP: do i = 1, 2
              if (signal_is_pending ()) return
              prt => shower%interactions(n_int)%i%partons(i)%p
              PARENT_PARTONS_LOOP: do
                 if (associated (prt%parent)) then
                    if (.not. parton_is_hadron (prt%parent)) then
                       prt => prt%parent
                    else
                       exit
                    end if
                 else
                    exit
                 end if
                 if (associated (prt%child2)) then
                    if (parton_is_branched (prt%child2)) then
                       call shower_parton_generate_fsr (shower, prt%child2)
                    end if
                 else
                    ! call msg_fatal ("Shower: no child2 associated?")
                 end if
              end do PARENT_PARTONS_LOOP
           end do INCOMING_PARTONS_LOOP
        end do INTERACTIONS_LOOP
```

```
      end subroutine shower_generate_fsr_for_partons_emitted_in_ISR
```

This executes the branching generated by `shower_generate_next_isr_branching`, that means it generates the flavours, momenta, etc.

⟨*Shower core: shower: TBP*⟩+≡
```
      procedure :: execute_next_isr_branching => shower_execute_next_isr_branching
```

⟨*Shower core: procedures*⟩+≡
```
      subroutine shower_execute_next_isr_branching (shower, prtp)
        class(shower_t), intent(inout) :: shower
        type(parton_pointer_t), intent(inout) :: prtp
        type(parton_t), pointer :: prt, otherprt
        type(parton_t), pointer :: prta, prtb, prtc, prtr
        real(default) :: mar, mbr
        real(default) :: phirand
        ! print *, "shower_execute_next_isr_branching"
        if (.not. associated (prtp%p)) then
            call msg_fatal ("Shower: prtp not associated")
        end if

        prt => prtp%p

        if ((.not. isr_pt_ordered .and. prt%t > - D_Min_t) .or. &
            (isr_pt_ordered .and. prt%scale < D_Min_scale)) then
            call msg_error ("Shower: no branching to be executed.")
        end if

        otherprt => shower_find_recoiler (shower, prt)
        if (isr_pt_ordered) then
           !!! get the recoiler
           otherprt => shower_find_recoiler (shower, prt)
           if (associated (otherprt%parent)) then
              !!! Why only for pt ordered
              if (.not. parton_is_hadron (otherprt%parent) .and. &
                  isr_pt_ordered) otherprt => otherprt%parent
           end if
           if (.not. associated (prt%parent)) then
              call shower%add_parent (prt)
           end if
           prt%parent%belongstoFSR = .false.
           if (.not. associated (prt%parent%child2)) then
              call shower%add_child (prt%parent, 2)
           end if
        end if

        prta => prt%parent          !!! new parton a with branching a->bc
        prtb => prt                 !!! former parton
        prtc => prt%parent%child2   !!! emitted parton
        prtr => otherprt            !!! recoiler

        mbr = (prtb%momentum + prtr%momentum)**1
        mar = mbr / sqrt(prt%z)

        !!! 1. assume you are in the restframe
        !!! 2. rotate by random phi
```

```
call tao_random_number (phirand)
phirand = twopi * phirand
call shower_apply_lorentztrafo (shower, &
     rotation(cos(phirand), sin(phirand),vector3_canonical(3)))
!!! 3. Put the b off-shell
!!! and
!!! 4. construct the massless a
!!! and the parton (eventually emitted by a)

!!! generate the flavour of the parent (prta)
if (prtb%aux_pt /= 0) prta%type = prtb%aux_pt
if (parton_is_quark (prtb)) then
   if (prta%type == prtb%type) then
      !!! (anti)-quark -> (anti-)quark + gluon
      prta%type = prtb%type   ! quarks have same flavour
      prtc%type = 21          ! emitted gluon
   else
      !!! gluon -> quark + antiquark
      prta%type = 21
      prtc%type = - prtb%type
   end if
else if (parton_is_gluon (prtb)) then
   prta%type = 21
   prtc%type = 21
else
   ! STOP "Bug in shower_execute_nexT_branching: neither quark nor gluon"
end if

prta%initial => prtb%initial
prta%belongstoFSR = .false.
prta%scale = prtb%scale
prta%x = prtb%x / prtb%z

prtb%momentum = vector4_moving ((mbr**2 + prtb%t) / (two * mbr), &
     vector3_canonical(3) * &
     sign ((mbr**2 - prtb%t) / (two * mbr), &
     vector4_get_component (prtb%momentum, 3)))
prtr%momentum = vector4_moving ((mbr**2 - prtb%t) / (two * mbr), &
     vector3_canonical(3) * &
     sign( (mbr**2 - prtb%t) / (two * mbr), &
     vector4_get_component(prtr%momentum, 3)))

prta%momentum = vector4_moving ((0.5_default / mbr) * &
     ((mbr**2 / prtb%z) + prtb%t - parton_mass_squared(prtc)), &
     vector3_null)
prta%momentum = vector4_moving (parton_get_energy (prta), &
     vector3_canonical(3) * &
     (0.5_default / vector4_get_component (prtb%momentum, 3)) * &
     ((mbr**2 / prtb%z) - two &
     * parton_get_energy(prtr) * parton_get_energy(prta) ) )
if (parton_get_energy(prta)**2 - vector4_get_component &
     (prta%momentum,3)**2 - parton_mass_squared (prtc) &
     > zero) then
   !!! This SHOULD be always fulfilled???
```

```
      prta%momentum = vector4_moving (parton_get_energy (prta), &
            vector3_moving([sqrt (parton_get_energy(prta)**2 - &
            vector4_get_component (prta%momentum, 3)**2 - &
            parton_mass_squared (prtc)), zero, &
            vector4_get_component(prta%momentum, 3) ]))
   end if
   prtc%momentum = prta%momentum - prtb%momentum

   !!! 5. rotate to have a along z-axis
   call shower_boost_to_CMframe (shower)
   call shower_rotate_to_z (shower)
   !!! 6. rotate back in phi
   call shower_apply_lorentztrafo (shower, rotation &
         (cos(-phirand), sin(-phirand), vector3_canonical(3)))
else
   if (prt%child2%t > parton_mass_squared(prt%child2)) then
      call shower_add_children_of_emitted_timelike_parton &
            (shower, prt%child2)
      call parton_set_simulated (prt%child2)
   end if

   call shower%add_parent (prt)
   call shower%add_child (prt%parent, 2)

   prt%parent%momentum = prt%momentum
   prt%parent%t = prt%t
   prt%parent%x = prt%x
   prt%parent%initial => prt%initial
   prt%parent%belongstoFSR = .false.

   prta => prt
   prtb => prt%child1
   prtc => prt%child2
end if
if (signal_is_pending ()) return
if (isr_pt_ordered) then
   call parton_generate_ps_ini (prt%parent)
else
   call parton_generate_ps_ini (prt)
end if

!!! add color connections
if (parton_is_quark (prtb)) then

   if (prta%type == prtb%type) then
      if (prtb%type > 0) then
         !!! quark -> quark + gluon
         prtc%c2 = prtb%c1
         prtc%c1 = shower%get_next_color_nr ()
         prta%c1 = prtc%c1
      else
         !!! antiquark -> antiquark + gluon
         prtc%c1 = prtb%c2
         prtc%c2 = shower%get_next_color_nr ()
```

```
               prta%c2 = prtc%c2
            end if
       else
          !!! gluon -> quark + antiquark
          if (prtb%type > 0) then
             !!! gluon -> quark + antiquark
             prta%c1 = prtb%c1
             prtc%c1 = 0
             prtc%c2 = shower%get_next_color_nr ()
             prta%c2 = prtc%c2
          else
             !!! gluon -> antiquark + quark
             prta%c2 = prtb%c2
             prtc%c1 = shower%get_next_color_nr ()
             prtc%c2 = 0
             prta%c1 = prtc%c1
          end if
       end if
    else if (parton_is_gluon (prtb)) then
       if (parton_is_gluon (prta)) then
          !!! g -> gg
          prtc%c2 = prtb%c1
          prtc%c1 = shower%get_next_color_nr ()
          prta%c1 = prtc%c1
          prta%c2 = prtb%c2
       else if (parton_is_quark (prta)) then
          if (prta%type > 0) then
             prta%c1 = prtb%c1
             prta%c2 = 0
             prtc%c1 = prtb%c2
             prtc%c2 = 0
          else
             prta%c1 = 0
             prta%c2 = prtb%c2
             prtc%c1 = 0
             prtc%c2 = prtb%c1
          end if
       end if
    else
       ! STOP "Bug in shower_execute_nexT_branching:
       !    neither quark nor gluon"
    end if

    call shower%sort_partons ()
    call shower_boost_to_CMframe (shower)
    call shower_rotate_to_z (shower)

    ! print *, "  shower_execute_next_isr_branching finished"
  end subroutine shower_execute_next_isr_branching
```

⟨*Shower core: procedures*⟩+≡
```
  subroutine shower_remove_parents_and_stuff (shower, prt)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout), target :: prt
```

```
      type(parton_t), pointer :: actprt, nextprt
      ! print *, " shower_remove_parents for parton ", prt%nr
      nextprt => prt%parent
      actprt => null()
      !!! remove children of emitted timelike parton
      if (associated (prt%child2)) then
         if (associated (prt%child2%child1)) then
            call shower_remove_parton_from_partons_recursive &
                 (shower, prt%child2%child1)
         end if
         prt%child2%child1 => null()
         if (associated (prt%child2%child2)) then
            call shower_remove_parton_from_partons_recursive &
                 (shower, prt%child2%child2)
         end if
         prt%child2%child2 => null()
      end if
      do
         actprt => nextprt
         if (.not. associated (actprt)) then
            exit
         else if (parton_is_hadron (actprt)) then
            !!! remove beam-remnant
            call shower_remove_parton_from_partons (shower, actprt%child2)
            exit
         end if
         if (associated (actprt%parent)) then
            nextprt => actprt%parent
         else
            nextprt => null()
         end if
         ! print *, " removing parton ", actprt%child2%nr
         call shower_remove_parton_from_partons_recursive &
              (shower, actprt%child2)
         ! print *, " removing parton ", actprt%nr
         call shower_remove_parton_from_partons (shower, actprt)

      end do
      prt%parent=>null()

      !      print *, " shower_remove_parents for parton finished"
    end subroutine shower_remove_parents_and_stuff
```

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: get_ISR_scale => shower_get_ISR_scale
```

⟨*Shower core: procedures*⟩+≡
```
  !!! MERGIND: not sure which function is the right one
  !!!    function shower_get_ISR_scale(shower) result (scale)
  !!!       class(shower_t), intent(in) :: shower
  !!!       real(default) :: scale
  !!!
  !!!       type(parton_t), pointer :: prt1, prt2
  !!!       integer :: i
```

2241

```
!!!
!!!       scale = (10._default)**10
!!!       do i = 1, size(shower%interactions)
!!!          call interaction_find_partons_nearest_to_hadron(shower%interactions(i)%i, &
!!!               prt1, prt2)
!!!          call shower%write ()
!!!          call parton_write(prt1)
!!!          call parton_write(prt2)
!!!
!!!          if (isr_pt_ordered) then
!!!             if ((parton_is_hadron(prt1%parent).eqv..false.).and.(abs(prt1%parent%scale).gt.scal
!!!             if ((parton_is_hadron(prt1%parent).eqv..false.).and.(abs(prt2%parent%scale).gt.scal
!!!          else
!!!             if ((parton_is_simulated(prt1)).and.(abs(prt1%t).lt.scale)) scale = abs(prt1%t)
!!!             if ((parton_is_simulated(prt2)).and.(abs(prt2%t).lt.scale)) scale = abs(prt2%t)
!!!!             if ((parton_is_hadron(prt1%parent).eqv..false.).and.(abs(prt2%parent%t).gt.scale))
!!!          end if
!!!       end do
!!!       print *, "returning: ", scale
!!!!       pause
!!!    end function shower_get_ISR_scale

  function shower_get_ISR_scale (shower) result (scale)
    class(shower_t), intent(in) :: shower
    real(default) :: scale
    type(parton_t), pointer :: prt1, prt2
    integer :: i
    scale = zero
    do i = 1, size (shower%interactions)
       call interaction_find_partons_nearest_to_hadron &
            (shower%interactions(i)%i, prt1, prt2)
       if (.not. parton_is_simulated (prt1) .and. abs(prt1%scale) > scale) &
            scale = abs(prt1%scale)
       if (.not. parton_is_simulated (prt1) .and. abs(prt2%scale) > scale) &
            scale = abs(prt2%scale)
    end do
    ! call shower%write ()
    ! call parton_write(prt1)
    ! call parton_write(prt2)
    ! print *, "returning: ", scale
  end function shower_get_ISR_scale
```

⟨*Shower core: shower: TBP*⟩+≡
```
    procedure :: set_max_isr_scale => shower_set_max_isr_scale
```

⟨*Shower core: procedures*⟩+≡
```
    subroutine shower_set_max_isr_scale (shower, newscale)
      class(shower_t), intent(inout) :: shower
      real(default), intent(in) :: newscale
      real(default) :: scale
      type(parton_t), pointer :: prt
      integer :: i,j
      print *, "shower_set_max_isr_scale", newscale
      call shower%write ()
```

```
            if (isr_pt_ordered) then
               scale = newscale
            else
               scale = -abs(newscale)
            end if

            INTERACTIONS: do i = 1, size (shower%interactions)
               PARTONS: do j = 1, 2
                  prt => shower%interactions(i)%i%partons(j)%p
                  do
                     if (.not. isr_pt_ordered) then
                        if (prt%belongstointeraction) prt => prt%parent
                     end if
                     if (prt%t < scale) then
                        if (associated (prt%parent)) then
                           prt => prt%parent
                        else
                           exit    !!! unresolved prt found
                        end if
                     else
                        exit    !!! prt with scale above newscale found
                     end if
                  end do
                  if (.not. isr_pt_ordered) then
                     if (prt%child1%belongstointeraction .or. &
                          parton_is_hadron (prt)) then
                        !!! don't reset scales of "first" spacelike partons
                        !!!    in virtuality ordered shower or hadrons
                        cycle
                     end if
                  else
                     if (parton_is_hadron (prt)) then
                        !!! don't reset scales of hadrons
                        cycle
                     end if
                  end if
                  if (isr_pt_ordered) then
                     prt%scale = scale
                  else
                     prt%t = scale
                  end if
                  call parton_set_simulated (prt, .false.)
                  call shower_remove_parents_and_stuff (shower, prt)
               end do PARTONS
            end do INTERACTIONS

            call shower%write ()
            print *, "shower_set_max_isr_scale finished"
         end subroutine shower_set_max_isr_scale
```

⟨*Shower core: shower: TBP*⟩+≡
```
      procedure :: interaction_generate_fsr_2ton => &
           shower_interaction_generate_fsr_2ton
```

⟨Shower core: procedures⟩+≡

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! new version
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!    subroutine shower_interaction_generate_fsr(shower, interaction)
!!!       type(shower_t), intent(inout) :: shower
!!!       type(shower_interaction_t), intent(inout) :: interaction
!!!       type(parton_pointer_t), dimension(:), allocatable :: partons   ! array of partons whose c
!!!
!!!       ! arrange partons to be included in <partons>
!!!       ! for qqbar state: include imaginary mother + first branching
!!!       call shower_prepare_for_simulate_fsr_ana(shower, interaction%out1%p, interaction%out2%p)
!!!
!!!       allocate(partons(1:1))
!!!       partons(1)%p => interaction%out1%p%child1
!!!       call shower_parton_pointer_array_generate_fsr(shower, partons)
!!!       call shower_parton_update_color_connections(shower, interaction%out1%p%child1%child1)
!!!       call shower_parton_update_color_connections(shower, interaction%out1%p%child1%child2)
!!!    end subroutine shower_interaction_generate_fsr

  subroutine shower_interaction_generate_fsr_2ton (shower, interaction)
    class(shower_t), intent(inout) :: shower
    type(shower_interaction_t), intent(inout) :: interaction
    type(parton_t), pointer :: prt
    prt => interaction%partons(3)%p
    do
       if (.not. associated (prt%parent)) exit
       prt => prt%parent
    end do
    call shower_parton_generate_fsr (shower, prt)
    call shower_parton_update_color_connections (shower, prt)
  end subroutine shower_interaction_generate_fsr_2ton
```

Perform the FSR for one parton, it is assumed, that the parton already branched. Hence, its children are to be simulated this procedure is intended for branched FSR-partons emitted in the ISR.

⟨Shower core: procedures⟩+≡

```
  subroutine shower_parton_generate_fsr (shower, prt)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout), target :: prt
    type(parton_pointer_t), dimension(:), allocatable :: partons
    if (signal_is_pending ()) return
    if (.not. parton_is_branched (prt)) then
       print *, " error in shower_parton_generate_fsr: parton not branched"
       return
    end if
    if (parton_is_simulated (prt%child1) .or. &
        parton_is_simulated (prt%child2)) then
       ! print *, " error in shower_parton_generate_fsr: children
       !        already simulated for parton ", prt%nr
       ! call shower%write ()
       return
```

```
         end if

         allocate (partons(1:1))
         partons(1)%p => prt
         call shower_parton_pointer_array_generate_fsr (shower, partons)

       end subroutine shower_parton_generate_fsr
```

⟨*Shower core: procedures*⟩+≡

```
       recursive subroutine shower_parton_pointer_array_generate_fsr &
             (shower, partons)
         type(shower_t), intent(inout) :: shower
         type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: &
             partons
         type(parton_pointer_t), dimension(:), allocatable :: partons_new
         integer :: i, size_partons, size_partons_new
         size_partons = size (partons)
         if (signal_is_pending ()) return
         if (size_partons == 0) return
         !!! sort partons -> necessary ?? -> probably not
         !!! simulate highest/first parton
         call shower_simulate_children_ana (shower, partons(1)%p)
         !!! check for new daughters to be included in new_partons
         size_partons_new = size_partons - 1    !!! partons(1) not needed anymore
         if (parton_is_branched (partons(1)%p%child1)) &
               size_partons_new = size_partons_new + 1
         if (parton_is_branched (partons(1)%p%child2)) &
               size_partons_new = size_partons_new + 1

         allocate (partons_new(1:size_partons_new))

         if (size_partons > 1) then
            do i = 2, size_partons
               partons_new(i - 1)%p => partons(i)%p
            end do
         end if
         if (parton_is_branched (partons(1)%p%child1)) &
               partons_new(size_partons)%p => partons(1)%p%child1
         if (parton_is_branched (partons(1)%p%child2)) then
            !!! check if child1 is already included
            if (size_partons_new == size_partons) then
               partons_new(size_partons)%p => partons(1)%p%child2
            else if (size_partons_new == size_partons + 1) then
               partons_new(size_partons + 1)%p => partons(1)%p%child2
            else
               call msg_fatal ("Shower: wrong sizes in" &
                     // "shower_parton_pointer_array_generate_fsr")
            end if
         end if
         deallocate(partons)

         ! print *, "end subroutine shower_parton_pointer_array_generate_fsr"
         call shower_parton_pointer_array_generate_fsr (shower, partons_new)
```

```
            end subroutine shower_parton_pointer_array_generate_fsr
```

⟨*Shower core: procedures*⟩+≡

```
    recursive subroutine shower_parton_update_color_connections &
        (shower, prt)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout) :: prt
    real(default) :: temprand
    if (.not. associated (prt%child1) .or. &
        .not. associated (prt%child2)) return

    if (signal_is_pending ()) return
    if (parton_is_gluon (prt)) then
        if (parton_is_quark (prt%child1)) then
            !!! give the quark the colorpartner and the antiquark
            !!!     the anticolorpartner
            if (prt%child1%type > 0) then
                !!! child1 is quark, child2 is antiquark
                prt%child1%c1 = prt%c1
                prt%child2%c2 = prt%c2
            else
                !!! child1 is antiquark, child2 is quark
                prt%child1%c2 = prt%c2
                prt%child2%c1 = prt%c1
            end if
        else
            !!! g -> gg splitting -> random choosing of partners
            call tao_random_number (temprand)
            if (temprand > 0.5_default) then
                prt%child1%c1 = prt%c1
                prt%child1%c2 = shower%get_next_color_nr ()
                prt%child2%c1 = prt%child1%c2
                prt%child2%c2 = prt%c2
            else
                prt%child1%c2 = prt%c2
                prt%child2%c1 = prt%c1
                prt%child2%c2 = shower%get_next_color_nr ()
                prt%child1%c1 = prt%child2%c2
            end if
        end if
    else if (parton_is_quark (prt)) then
        if (parton_is_quark (prt%child1)) then
            if (prt%child1%type > 0) then
                !!! q -> q + g
                prt%child2%c1 = prt%c1
                prt%child2%c2 = shower%get_next_color_nr ()
                prt%child1%c1 = prt%child2%c2
            else
                !!! qbar -> qbar + g
                prt%child2%c2 = prt%c2
                prt%child2%c1 = shower%get_next_color_nr ()
                prt%child1%c2 = prt%child2%c1
            end if
        else
```

```
            if (prt%child2%type > 0) then
                !!! q -> g + q
                prt%child1%c1 = prt%c1
                prt%child1%c2 = shower%get_next_color_nr ()
                prt%child2%c1 = prt%child1%c2
            else
                !!! qbar -> g + qbar
                prt%child1%c2 = prt%c2
                prt%child1%c1 = shower%get_next_color_nr ()
                prt%child2%c2 = prt%child1%c1
            end if
        end if
    end if

    call shower_parton_update_color_connections (shower, prt%child1)
    call shower_parton_update_color_connections (shower, prt%child2)
end subroutine shower_parton_update_color_connections
```

The next two routines are for PDFs. Wrapper function to return parton densities. Note that the arguments for evolvepdf need to be defined as double explicitly.

⟨*Shower core: procedures*⟩+≡

```
function get_pdf (mother, x, Q2, daughter) result (pdf)
    !    type(shower_t), intent(in), pointer :: shower
    integer, intent(in) :: mother, daughter
    real(default), intent(in) :: x, Q2
    real(default) :: pdf
    real(double), save :: f(-6:6) = 0._double
    real(double), save :: lastx, lastQ2 = 0._double

    if (abs (mother) /= 2212) then
        call msg_fatal ("Shower: pdf only implemented for (anti-)proton")
    else
        if (x > zero .and. x < one) then
            if (dble(Q2) /= lastQ2 .or. dble(x) /= lastx) then
                ! call evolvePDF(DBLE(x),sqrt(abs(DBLE(Q2))),f)  !! LHAPDF
                call shower_pdf_func &
                        (shower_pdf_set, dble(x), sqrt (abs (dble(Q2))), f)
            end if
            if (abs (daughter) >= 1 .and. abs (daughter) <= 6) then
                pdf = max (f(daughter * sign (1,mother)), 1E-10_default) / x
            else if (daughter == 21) then
                pdf = max (f(0), 1E-10_default) / x
            else
                print *, "error in pdf, unknown daughter", daughter
                pdf = zero
            end if
        else
            pdf = zero
        end if
    end if
    lastQ2 = dble(Q2)
    lastx  = dble(x)
```

```
      end function get_pdf
```

Wrapper function to return momentum densities. The arguments for `evolvepdf` need to be defined as `double` explicitly.

⟨*Shower core: procedures*⟩+≡
```
    function get_xpdf (mother, x, Q2, daughter) result (pdf)
      !   type(shower_t), intent(in), pointer :: shower
      integer, intent(in) :: mother, daughter
      real(default), intent(in) :: x, Q2
      real(default) :: pdf
      real(double), save :: f(-6:6) = 0._double
      real(double), save :: lastx, lastQ2 =0._double

      if (abs (mother) /= 2212) then
         call msg_fatal ("Shower: pdf only implemented for (anti-)proton")
      else
         if (x > zero .and. x < one) then
            if (dble(Q2) /= lastQ2 .or. dble(x) /= lastx) then
               ! call evolvePDF(DBLE(x),sqrt(abs(DBLE(Q2))),f)  !! LHAPDF
               call shower_pdf_func &
                    (shower_pdf_set, dble(x), sqrt (abs (dble(Q2))), f)
            end if
            if (abs(daughter) >= 1 .and. abs(daughter) <= 6) then
               pdf = max (f (daughter * sign (1,mother)), 1E-10_default)
            else if (daughter == 21) then
               pdf = max(f(0), 1E-10_default)
            else
               print *, "error in pdf, unknown daughter", daughter
               pdf = zero
            end if
         else
            pdf = zero
         end if
      end if
      lastQ2 = dble(Q2)
      lastx  = dble(x)
    end function get_xpdf
```

16.4 Talking to PYTHIA

⟨shower_topythia.f90⟩≡
  ⟨*File header*⟩

  module shower_topythia

  ⟨*Use kinds*⟩
    use lorentz !NODEP!
    use shower_base
    use shower_partons
    use shower_core

```

*⟨Standard module head⟩*

*⟨Shower2pythia: public⟩*

```
contains
```

⟨Shower2pythia: procedures⟩

```
      end module shower_topythia
```

⟨Shower2pythia: public⟩≡
```
      public :: shower_converttopythia
```

⟨Shower2pythia: procedures⟩≡
```
      subroutine shower_converttopythia (shower)
        IMPLICIT DOUBLE PRECISION(A-H, O-Z)
        IMPLICIT INTEGER(I-N)
        !!!   C... Commonblocks.
        COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
        COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
        COMMON/PYDAT2/KCHG(500,4),PMAS(500,4),PARF(2000),VCKM(4,4)
        COMMON/PYDAT3/MDCY(500,3),MDME(8000,2),BRAT(8000),KFDP(8000,5)
        COMMON/PYSUBS/MSEL,MSELPD,MSUB(500),KFIN(2,-40:40),CKIN(200)
        COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
        SAVE /PYJETS/,/PYDAT1/,/PYDAT2/,/PYDAT3/,/PYSUBS/,/PYPARS/

        type(shower_t), intent(in) :: shower
        type(parton_t), pointer :: pp, ppparent
        integer :: i, j, nz

        !!! currently only works for one interaction

        do i = 1, 2
           !!! get history of the event
           pp => shower%interactions(1)%i%partons(i)%p
           !!! add these partons to the event record
           if (associated (pp%initial)) then
              !!! add hadrons
              K(i,1) = 21
              K(i,2) = pp%initial%type
              K(i,3) = 0
              P(i,1) = vector4_get_component (pp%initial%momentum,1)
              P(i,2) = vector4_get_component (pp%initial%momentum,2)
              P(i,3) = vector4_get_component (pp%initial%momentum,3)
              P(i,4) = vector4_get_component (pp%initial%momentum,0)
              P(I,5) = pp%initial%momentum**2
              !!! add partons emitted by the hadron
              ppparent => pp
              do while (associated (ppparent%parent))
                 if (parton_is_hadron (ppparent%parent)) then
                    exit
                 else
                    ppparent => ppparent%parent
                 end if
```

```
      end do
      K(i+2,1) = 21
      K(i+2,2) = ppparent%type
      K(i+2,3) = i
      P(i+2,1) = vector4_get_component (ppparent%momentum,1)
      P(i+2,2) = vector4_get_component (ppparent%momentum,2)
      P(i+2,3) = vector4_get_component (ppparent%momentum,3)
      P(i+2,4) = vector4_get_component (ppparent%momentum,0)
      P(I+2,5) = ppparent%momentum**2
      !!! add partons in the initial state of the ME
      K(i+4,1) = 21
      K(i+4,2) = pp%type
      K(i+4,3) = i
      P(i+4,1) = vector4_get_component (pp%momentum,1)
      P(i+4,2) = vector4_get_component (pp%momentum,2)
      P(i+4,3) = vector4_get_component (pp%momentum,3)
      P(i+4,4) = vector4_get_component (pp%momentum,0)
      P(I+4,5) = pp%momentum**2
   else
      !!! for e+e- without ISR all entries are the same
      K(i,1) = 21
      K(i,2) = pp%type
      K(i,3) = 0
      P(i,1) = vector4_get_component (pp%momentum,1)
      P(i,2) = vector4_get_component (pp%momentum,2)
      P(i,3) = vector4_get_component (pp%momentum,3)
      P(i,4) = vector4_get_component (pp%momentum,0)
      P(I,5) = pp%momentum**2
      do j = 1, 5
         P(i+2,j) = P(1,j)
         K(i+2,j) = K(1,j)
         K(i+2,3) = i
         P(i+4,j) = P(1,j)
         K(i+4,j) = K(1,j)
         K(i+4,3) = i
      end do
      P(i+4,5) = 0.
   end if
end do
N = 6
!!! create intermediate (fake) Z-Boson
K(7,1) = 21
K(7,2) = 23
K(7,3) = 0
P(7,1) = P(5,1) + P(6,1)
P(7,2) = P(5,2) + P(6,2)
P(7,3) = P(5,3) + P(6,3)
P(7,4) = P(5,4) + P(6,4)
P(7,5) = P(7,4)**2 - P(7,3)**2 - P(7,2)**2 - P(7,1)**2
N = 7
!!! include partons in the final state of the hard matrix element
do i = 1, size (shower%interactions(1)%i%partons) - 2
   !!! get partons that are in the final state of the hard matrix element
   pp => shower%interactions(1)%i%partons(2+i)%p
```

```
      !!! add these partons to the event record
      K(7+I,1) = 21
      K(7+I,2) = pp%type
      K(7+I,3) = 7
      P(7+I,1) = vector4_get_component (pp%momentum, 1)
      P(7+I,2) = vector4_get_component (pp%momentum, 2)
      P(7+I,3) = vector4_get_component (pp%momentum, 3)
      P(7+I,4) = vector4_get_component (pp%momentum, 0)
      P(7+I,5) = P(7+I,4)**2 - P(7+I,3)**2 - P(7+I,2)**2 - P(7+I,1)**2
      N = 7 + I
    end do
    !!! include "Z" (again)
    N = N + 1
    K(N,1) = 11
    K(N,2) = 23
    K(N,3) = 7
    P(N,1) = P(7,1)
    P(N,2) = P(7,2)
    P(N,3) = P(7,3)
    P(N,4) = P(7,4)
    P(N,5) = P(7,5)
    nz = N
    !!! include partons from the final state of the parton shower
    call shower_transfer_final_partons_to_pythia (shower, 8)
    !!! set "children" of "Z"
    K(nz,4) = 11
    K(nz,5) = N
    !!! mark spacers
    MSTU(73) = N
    MSTU(74) = N

    !!! be sure to remove the next partons (=first obsolete partons)
    K(N+1,1) = 0
    K(N+1,2) = 0
    K(N+1,3) = 0
    K(N+2,1) = 0
    K(N+2,2) = 0
    K(N+2,3) = 0
    K(N+3,1) = 0
    K(N+3,2) = 0
    K(N+3,3) = 0
    !!! otherwise they might be interpreted as thrust information
  end subroutine shower_converttopythia
```

⟨*Shower2pythia: procedures*⟩+≡

```
    subroutine shower_transfer_final_partons_to_pythia (shower, first)
      IMPLICIT DOUBLE PRECISION(A-H, O-Z)
      IMPLICIT INTEGER(I-N)
      !    C... Commonblocks.
      COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
      COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
      COMMON/PYDAT2/KCHG(500,4),PMAS(500,4),PARF(2000),VCKM(4,4)
      COMMON/PYDAT3/MDCY(500,3),MDME(8000,2),BRAT(8000),KFDP(8000,5)
      COMMON/PYSUBS/MSEL,MSELPD,MSUB(500),KFIN(2,-40:40),CKIN(200)
```

```
      COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
      SAVE /PYJETS/,/PYDAT1/,/PYDAT2/,/PYDAT3/,/PYSUBS/,/PYPARS/

      type(shower_t), intent(in) :: shower
      integer, intent(in) :: first
      type(parton_t), pointer :: prt
      integer :: i, j, n_finals
      type(parton_t), dimension(:), allocatable :: final_partons
      type(parton_t) :: temp_parton
      integer :: minindex, maxindex


      prt => null()

!!! !!! old version using (anti-)colorpartner pointers
!!!
!!!      partons: do i = 1, size(shower%partons)
!!!         ! loop over partons to find quarks = beginnings of color strings
!!!         if (.not. associated(shower%partons(i)%p)) cycle
!!!         prt=> shower%partons(i)%p
!!!         if (associated(prt%child1)) cycle
!!!         if (.not. parton_is_quark(prt)) cycle
!!!         if (.not. prt%type > 0 ) cycle
!!!
!!!         color_string: do
!!!             ! transfer prt to PYHIA
!!!             N = N+1
!!!             K(N,1) = 2
!!!             K(N,2) = prt%type
!!!             K(N,3) = first
!!!             K(N,4) = 0
!!!             K(N,5) = 0
!!!             P(N,1) = vector4_get_component(prt%momentum, 1)
!!!             P(N,2) = vector4_get_component(prt%momentum, 2)
!!!             P(N,3) = vector4_get_component(prt%momentum, 3)
!!!             P(N,4) = vector4_get_component(prt%momentum, 0)
!!!             P(N,5) = prt%t
!!!
!!!             if (associated(prt%colorpartner)) then
!!!                 prt=>prt%colorpartner
!!!                 cycle color_string
!!!             else
!!!                 K(N,1) = 1 !mark end of string
!!!                 exit color_string
!!!             end if
!!!         end do color_string
!!!      end do partons

!!! new version using color indices

!!! get total number of final partons
n_finals = 0
do i = 1, size (shower%partons)
   if (.not. associated (shower%partons(i)%p)) cycle
   prt => shower%partons(i)%p


                              2252
```

```
      if (.not. prt%belongstoFSR) cycle
      if (associated (prt%child1)) cycle
      n_finals = n_finals + 1
   end do
!!! print *, "n_finals=", n_finals

   allocate (final_partons(1:n_finals))
   j = 1
   do i = 1, size (shower%partons)
      if (.not. associated (shower%partons(i)%p)) cycle
      prt => shower%partons(i)%p
      if (.not. prt%belongstoFSR) cycle
      if (associated (prt%child1)) cycle
      final_partons(j) = shower%partons(i)%p
      j = j + 1
   end do

!!! move quark to front as beginning of color string
   minindex = 1
   maxindex = size (final_partons)
   FIND_Q: do i = minindex, maxindex
      if (final_partons(i)%type >= 1 .and. final_partons(i)%type <= 6) then
         temp_parton = final_partons(minindex)
         final_partons(minindex) = final_partons(i)
         final_partons(i) = temp_parton
         exit FIND_Q
      end if
   end do FIND_Q

!!! sort so that connected partons are next to each other, don't care about zeros
   do i = 1, size (final_partons)
      !!! ensure that final_partnons begins with a color (not an anticolor)
      if (final_partons(i)%c1 > 0 .and. final_partons(i)%c2 == 0) then
         if (i == 1) then
            exit
         else
            temp_parton = final_partons(1)
            final_partons(1) = final_partons(i)
            final_partons(i) = temp_parton
            exit
         end if
      end if
   end do

   do i = 1, size (final_partons) - 1
      !!! search for color partner and move it to i + 1
      PARTNERS: do j = i + 1, size (final_partons)
         if (final_partons(j)%c2 == final_partons(i)%c1) exit PARTNERS
      end do PARTNERS
      if (j > size (final_partons)) then
         print *, "no color connected parton found" !WRONG???
         print *, "particle: ", final_partons(i)%nr, " index: ", &
               final_partons(i)%c1
         exit
```

```
        end if
        temp_parton = final_partons(i + 1)
        final_partons(i + 1) = final_partons(j)
        final_partons(j) = temp_parton
     end do

     !!! transfering partons
     do i = 1, size (final_partons)
        prt = final_partons(i)
        N = N + 1
        K(N,1) = 2
        if (prt%c1 == 0) K(N,1) = 1        !!! end of color string
        K(N,2) = prt%type
        K(N,3) = first
        K(N,4) = 0
        K(N,5) = 0
        P(N,1) = vector4_get_component (prt%momentum, 1)
        P(N,2) = vector4_get_component (prt%momentum, 2)
        P(N,3) = vector4_get_component (prt%momentum, 3)
        P(N,4) = vector4_get_component (prt%momentum, 0)
        P(N,5) = prt%momentum**2
     end do
     deallocate (final_partons)
   end subroutine shower_transfer_final_partons_to_pythia

⟨Shower2pythia: procedures⟩+≡
   recursive subroutine shower_topythia_recursive_weighted (prt, mode, first)
     IMPLICIT DOUBLE PRECISION(A-H, O-Z)
     IMPLICIT INTEGER(I-N)
     !   C... Commonblocks.
     COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
     COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
     COMMON/PYDAT2/KCHG(500,4),PMAS(500,4),PARF(2000),VCKM(4,4)
     COMMON/PYDAT3/MDCY(500,3),MDME(8000,2),BRAT(8000),KFDP(8000,5)
     COMMON/PYSUBS/MSEL,MSELPD,MSUB(500),KFIN(2,-40:40),CKIN(200)
     COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
     SAVE /PYJETS/,/PYDAT1/,/PYDAT2/,/PYDAT3/,/PYSUBS/,/PYPARS/

     type(parton_t), intent(in), target :: prt
     integer, intent(in) :: mode, first
     integer :: n_emissions
     type(parton_t), pointer :: tempprt, finalprt
     real(default), dimension(:), allocatable :: costhetas
     type(parton_pointer_t), dimension(:), allocatable :: emittedpartons
     integer :: i, max
     real(default) :: maxcostheta

     n_emissions = 0

     if (parton_is_final(prt)) then
        N = N + 1
        K(N,1) = 2
        if (parton_is_quark(prt)) then
           ! check if quark is end of a color connection
```

2254

```
            if (prt%type < 0) then
               K(N,1) = 1
            end if
         end if
         K(N,2) = prt%type
         K(N,3) = first
         K(N,4) = 0
         K(N,5) = 0
         P(N,1) = vector4_get_component (prt%momentum, 1)
         P(N,2) = vector4_get_component (prt%momentum, 2)
         P(N,3) = vector4_get_component (prt%momentum, 3)
         P(N,4) = vector4_get_component (prt%momentum, 0)
         P(N,5) = prt%t
      else
         !!! search for following final partons
         if (parton_is_gluon (prt)) then
            if (parton_is_gluon(prt%child1)) then
               !!! g-> gg so sequence is unimportant
               call shower_topythia_recursive_weighted (prt%child1, 1, first)
               call shower_topythia_recursive_weighted (prt%child2, 1, first)
            else
               !!! g-> qqbar -> use antiquark first,
               !!!        so that color flow is given correctly
               if (prt%child1%type < 0) then
                  call shower_topythia_recursive_weighted (prt%child1, 1, first)
                  call shower_topythia_recursive_weighted (prt%child2, 2, first)
               else
                  call shower_topythia_recursive_weighted (prt%child2, 2, first)
                  call shower_topythia_recursive_weighted (prt%child1, 1, first)
               end if
            end if
         else
            !!! parton is quark
            !!! find the emitted gluons and order them by the emission angle
            n_emissions = 0
            tempprt = prt
            do
               !!! calculate how many emissions there are
               if (associated (tempprt%child1)) then
                  tempprt => tempprt%child1
                  n_emissions = n_emissions +1
                  cycle
               else
                  exit
               end if
            end do

            allocate (costhetas(1:n_emissions))
            allocate (emittedpartons(1:n_emissions))

            tempprt => prt
            n_emissions = 1
            do
               if (associated (tempprt%child1)) then
```

```fortran
            costhetas(n_emissions) = parton_get_costheta_correct (tempprt)
            emittedpartons(n_emissions)%p => tempprt%child2
            n_emissions = n_emissions +1
            tempprt => tempprt%child1
            cycle
         else
            finalprt=>tempprt
            exit
         end if
      end do

      !!! if mode .eq. 1 write quark first
      if (mode == 1) &
           call shower_topythia_recursive_weighted (finalprt, 1, first)

      !!! if mode .eq. 2 write out gluons in recursive order <= replace
      !!!      costheta by 1- costheta
      if (mode == 2) then
         do i = 1, size (costhetas)
            costhetas(i) = one - costhetas(i)
         end do
      end if

      do
         max = 0
         maxcostheta = zero
         do i = 1, size (costhetas)
            if (costhetas(i) > maxcostheta) then
               maxcostheta = costhetas(i)
               max = i
            end if
         end do
         if (maxcostheta == zero) then
            exit
         end if

         call shower_topythia_recursive_weighted &
              (emittedpartons(max)%p, mode, first)
         costhetas(max) = zero
      end do

      !!! if mode .eq. 2 write quark last
      if (mode .eq. 2) then
         call shower_topythia_recursive_weighted (finalprt, 2, first)
      end if

      ! if (mode == 1) then
      !    call shower_topythia_recursive_weighted (prt%child1, 1, first)
      !    call shower_topythia_recursive_weighted (prt%child2, 1, first)
      ! else
      !    call shower_topythia_recursive_weighted (prt%child2, 2, first)
      !    call shower_topythia_recursive_weighted (prt%child1, 2, first)
      ! end if
   end if
```

```
          end if
    end subroutine shower_topythia_recursive_weighted


⟨Shower2pythia: procedures⟩+≡
    recursive subroutine shower_topythia_recursive (prt, mode,first)
      IMPLICIT DOUBLE PRECISION(A-H, O-Z)
      IMPLICIT INTEGER(I-N)
      !   C... Commonblocks.
      COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
      COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
      COMMON/PYDAT2/KCHG(500,4),PMAS(500,4),PARF(2000),VCKM(4,4)
      COMMON/PYDAT3/MDCY(500,3),MDME(8000,2),BRAT(8000),KFDP(8000,5)
      COMMON/PYSUBS/MSEL,MSELPD,MSUB(500),KFIN(2,-40:40),CKIN(200)
      COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
      SAVE /PYJETS/,/PYDAT1/,/PYDAT2/,/PYDAT3/,/PYSUBS/,/PYPARS/

      type(parton_t), intent(in) :: prt
      integer, intent(in) :: mode, first

      if (parton_is_final (prt)) then
         N = N + 1
         K(N,1) = 2
         if (parton_is_quark(prt)) then
            !!! check if quark is end of a color connection
            if (prt%type < 0) then
               K(N,1) = 1
            end if
         end if
         K(N,2) = prt%type
         K(N,3) = first
         K(N,4) = 0
         K(N,5) = 0
         P(N,1) = vector4_get_component (prt%momentum, 1)
         P(N,2) = vector4_get_component (prt%momentum, 2)
         P(N,3) = vector4_get_component (prt%momentum, 3)
         P(N,4) = vector4_get_component (prt%momentum, 0)
         P(N,5) = prt%t
      else
         !!! search for following final partons
         if (parton_is_gluon (prt)) then
            if (parton_is_gluon (prt%child1)) then
               !!! g-> gg so sequence is unimportant
               call shower_topythia_recursive (prt%child1, 1, first)
               call shower_topythia_recursive (prt%child2, 1, first)
            else
               !!! g-> qqbar -> use antiquark first, so that color flow
               !!!     is given correctly
               if (prt%child1%type < 0) then
                  call shower_topythia_recursive (prt%child1, 1, first)
                  call shower_topythia_recursive (prt%child2, 2, first)
               else
                  call shower_topythia_recursive (prt%child2, 2, first)
                  call shower_topythia_recursive (prt%child1, 1, first)
               end if
            end if
```

```
            endif
        else
            if (mode == 1) then
                call shower_topythia_recursive (prt%child1, 1, first)
                call shower_topythia_recursive (prt%child2, 1, first)
            else
                call shower_topythia_recursive (prt%child2, 2, first)
                call shower_topythia_recursive (prt%child1, 2, first)
            end if
        end if
    end if
  end subroutine shower_topythia_recursive
```

⟨pythia_up.f⟩≡
```
  C*********************************************************************
  C*********************************************************************
  C*                                                                 **
  C*                                                    Mar 2011      **
  C*                                                                 **
  C*                        The Lund Monte Carlo                      **
  C*                                                                 **
  C*                        PYTHIA version 6.4                        **
  C*                                                                 **
  C*                        Torbjorn Sjostrand                        **
  C*                  Department of Theoretical Physics               **
  C*                          Lund University                         **
  C*                Solvegatan 14A, S-223 62 Lund, Sweden             **
  C*                    E-mail torbjorn@thep.lu.se                    **
  C*                                                                 **
  C*                    SUSY and Technicolor parts by                 **
  C*                          Stephen Mrenna                          **
  C*                          Computing Division                      **
  C*              Generators and Detector Simulation Group            **
  C*              Fermi National Accelerator Laboratory               **
  C*                   MS 234, Batavia, IL  60510, USA                **
  C*                    phone + 1 - 630 - 840 - 2556                  **
  C*                      E-mail mrenna@fnal.gov                      **
  C*                                                                 **
  C*          New multiple interactions and more SUSY parts by        **
  C*                          Peter Skands                            **
  C*                CERN/PH, CH-1211 Geneva, Switzerland              **
  C*                     phone +41 - 22 - 767 2447                    **
  C*                     E-mail peter.skands@cern.ch                  **
  C*                                                                 **
  C*          Several parts are written by Hans-Uno Bengtsson         **
  C*           PYSHOW is written together with Mats Bengtsson         **
  C*                 PYMAEL is written by Emanuel Norrbin             **
  C*      advanced popcorn baryon production written by Patrik Eden   **
  C*      code for virtual photons mainly written by Christer Friberg **
  C*      code for low-mass strings mainly written by Emanuel Norrbin **
  C*          Bose-Einstein code mainly written by Leif Lonnblad      **
  C*        CTEQ  parton distributions are by the CTEQ collaboration  **
  C*          GRV 94 parton distributions are by Glueck, Reya and Vogt **
  C*      SaS photon parton distributions together with Gerhard Schuler **
```

2258

```
C*       g + g and q + qbar -> t + tbar + H code by Zoltan Kunszt      **
C*          MSSM Higgs mass calculation code by M. Carena,             **
C*           J.R. Espinosa, M. Quiros and C.E.M. Wagner                **
C* UED implementation by M. Elkacimi, D. Goujdami, H. Przysiezniak **
C*          PYGAUS adapted from CERN library (K.S. Kolbig)             **
C*          NRQCD/colour octet production of onium by S. Wolf          **
C*                                                                     **
C*   The latest program version and documentation is found on WWW      **
C*             http://www.thep.lu.se/~torbjorn/Pythia.html             **
C*                                                                     **
C*              Copyright Torbjorn Sjostrand, Lund 2010                **
C*                                                                     **
C*********************************************************************
C*********************************************************************

C...UPINIT
C...Dummy routine, to be replaced by a user implementing external
C...processes. Is supposed to fill the HEPRUP commonblock with info
C...on incoming beams and allowed processes.

C...New example: handles a standard Les Houches Events File.

      SUBROUTINE UPINIT

C...Double precision and integer declarations.
      IMPLICIT DOUBLE PRECISION(A-H, O-Z)
      IMPLICIT INTEGER(I-N)

C...PYTHIA commonblock: only used to provide read unit MSTP(161).
      COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
      SAVE /PYPARS/

C...User process initialization commonblock.
      INTEGER MAXPUP
      PARAMETER (MAXPUP=100)
      INTEGER IDBMUP,PDFGUP,PDFSUP,IDWTUP,NPRUP,LPRUP
      DOUBLE PRECISION EBMUP,XSECUP,XERRUP,XMAXUP
      COMMON/HEPRUP/IDBMUP(2),EBMUP(2),PDFGUP(2),PDFSUP(2),
     &IDWTUP,NPRUP,XSECUP(MAXPUP),XERRUP(MAXPUP),XMAXUP(MAXPUP),
     &LPRUP(MAXPUP)
      SAVE /HEPRUP/

C...Lines to read in assumed never longer than 200 characters.
      PARAMETER (MAXLEN=200)
      CHARACTER*(MAXLEN) STRING

C...Format for reading lines.
      CHARACTER*6 STRFMT
      STRFMT='(A000)'
      WRITE(STRFMT(3:5),'(I3)') MAXLEN

C...Loop until finds line beginning with "<init>" or "<init ".
  100 READ(MSTP(161),STRFMT,END=130,ERR=130) STRING
      IBEG=0
```

```
       110 IBEG=IBEG+1
C...Allow indentation.
           IF(STRING(IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-5) GOTO 110
           IF(STRING(IBEG:IBEG+5).NE.'<init>'.AND.
          &STRING(IBEG:IBEG+5).NE.'<init ') GOTO 100

C...Read first line of initialization info.
           READ(MSTP(161),*,END=130,ERR=130) IDBMUP(1),IDBMUP(2),EBMUP(1),
          &EBMUP(2),PDFGUP(1),PDFGUP(2),PDFSUP(1),PDFSUP(2),IDWTUP,NPRUP

C...Read NPRUP subsequent lines with information on each process.
           DO 120 IPR=1,NPRUP
             READ(MSTP(161),*,END=130,ERR=130) XSECUP(IPR),XERRUP(IPR),
          &   XMAXUP(IPR),LPRUP(IPR)
       120 CONTINUE
           RETURN

C...Error exit: give up if initalization does not work.
       130 WRITE(*,*) ' Failed to read LHEF initialization information.'
           WRITE(*,*) ' Event generation will be stopped.'
           CALL PYSTOP(12)

           RETURN
           END

C...Old example: handles a simple Pythia 6.4 initialization file.

c           SUBROUTINE UPINIT

C...Double precision and integer declarations.
c           IMPLICIT DOUBLE PRECISION(A-H, O-Z)
c           IMPLICIT INTEGER(I-N)

C...Commonblocks.
c           COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
c           COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
c           SAVE /PYDAT1/,/PYPARS/

C...User process initialization commonblock.
c           INTEGER MAXPUP
c           PARAMETER (MAXPUP=100)
c           INTEGER IDBMUP,PDFGUP,PDFSUP,IDWTUP,NPRUP,LPRUP
c           DOUBLE PRECISION EBMUP,XSECUP,XERRUP,XMAXUP
c           COMMON/HEPRUP/IDBMUP(2),EBMUP(2),PDFGUP(2),PDFSUP(2),
c          &IDWTUP,NPRUP,XSECUP(MAXPUP),XERRUP(MAXPUP),XMAXUP(MAXPUP),
c          &LPRUP(MAXPUP)
c           SAVE /HEPRUP/

C...Read info from file.
c           IF(MSTP(161).GT.0) THEN
c             READ(MSTP(161),*,END=110,ERR=110) IDBMUP(1),IDBMUP(2),EBMUP(1),
c          &   EBMUP(2),PDFGUP(1),PDFGUP(2),PDFSUP(1),PDFSUP(2),IDWTUP,NPRUP
c             DO 100 IPR=1,NPRUP
c               READ(MSTP(161),*,END=110,ERR=110) XSECUP(IPR),XERRUP(IPR),
```

```
c     &    XMAXUP(IPR),LPRUP(IPR)
c 100  CONTINUE
c        RETURN
C...Error or prematurely reached end of file.
c 110  WRITE(MSTU(11),5000)
c        STOP

C...Else not implemented.
c     ELSE
c        WRITE(MSTU(11),5100)
c        STOP
c     ENDIF

C...Format for error printout.
c 5000 FORMAT(1X,'Error: UPINIT routine failed to read information'/
c    &1X,'Execution stopped!')
c 5100 FORMAT(1X,'Error: You have not implemented UPINIT routine'/
c    &1X,'Dummy routine in PYTHIA file called instead.'/
c    &1X,'Execution stopped!')

c     RETURN
c     END


C**********************************************************************

C...UPEVNT
C...Dummy routine, to be replaced by a user implementing external
C...processes. Depending on cross section model chosen, it either has
C...to generate a process of the type IDPRUP requested, or pick a type
C...itself and generate this event. The event is to be stored in the
C...HEPEUP commonblock, including (often) an event weight.

C...New example: handles a standard Les Houches Events File.

      SUBROUTINE UPEVNT

C...Double precision and integer declarations.
      IMPLICIT DOUBLE PRECISION(A-H, O-Z)
      IMPLICIT INTEGER(I-N)

C...PYTHIA commonblock: only used to provide read unit MSTP(162).
      COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
      SAVE /PYPARS/

C...Added by WHIZARD
      COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
      SAVE/PYDAT1/

C...User process event common block.
      INTEGER MAXNUP
      PARAMETER (MAXNUP=500)
      INTEGER NUP,IDPRUP,IDUP,ISTUP,MOTHUP,ICOLUP
      DOUBLE PRECISION XWGTUP,SCALUP,AQEDUP,AQCDUP,PUP,VTIMUP,SPINUP
      COMMON/HEPEUP/NUP,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP,IDUP(MAXNUP),
```

2261

```
     &ISTUP(MAXNUP),MOTHUP(2,MAXNUP),ICOLUP(2,MAXNUP),PUP(5,MAXNUP),
     &VTIMUP(MAXNUP),SPINUP(MAXNUP)
      SAVE /HEPEUP/

C...Lines to read in assumed never longer than 200 characters.
      PARAMETER (MAXLEN=200)
      CHARACTER*(MAXLEN) STRING

C...Format for reading lines.
      CHARACTER*6 STRFMT
      STRFMT='(A000)'
      WRITE(STRFMT(3:5),'(I3)') MAXLEN

C...Loop until finds line beginning with "<event>" or "<event ".
  100 READ(MSTP(162),STRFMT,END=130,ERR=130) STRING
      IBEG=0
  110 IBEG=IBEG+1
C...Allow indentation.
      IF(STRING(IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-6) GOTO 110
      IF(STRING(IBEG:IBEG+6).NE.'<event>'.AND.
     &STRING(IBEG:IBEG+6).NE.'<event ') GOTO 100

C...Read first line of event info.
      READ(MSTP(162),*,END=130,ERR=130) NUP,IDPRUP,XWGTUP,SCALUP,
     &AQEDUP,AQCDUP

C...Read NUP subsequent lines with information on each particle.
      DO 120 I=1,NUP
        READ(MSTP(162),*,END=130,ERR=130) IDUP(I),ISTUP(I),
     &  MOTHUP(1,I),MOTHUP(2,I),ICOLUP(1,I),ICOLUP(2,I),
     &  (PUP(J,I),J=1,5),VTIMUP(I),SPINUP(I)
  120 CONTINUE
      RETURN

C...Error exit, typically when no more events.
  130 CONTINUE
C     WRITE(*,*) ' Failed to read LHEF event information.'
C     WRITE(*,*) ' Will assume end of file has been reached.'
      NUP=0
      MSTI(51)=1
C...Added by WHIZARD, mark these failed events
      MSTU(23)=1

      RETURN
      END

C...Old example: handles a simple Pythia 6.4 event file.


c     SUBROUTINE UPEVNT

C...Double precision and integer declarations.
c     IMPLICIT DOUBLE PRECISION(A-H, O-Z)
c     IMPLICIT INTEGER(I-N)
```

```
C...Commonblocks.
c       COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
c       COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
c       SAVE /PYDAT1/,/PYPARS/

C...User process event common block.
c       INTEGER MAXNUP
c       PARAMETER (MAXNUP=500)
c       INTEGER NUP,IDPRUP,IDUP,ISTUP,MOTHUP,ICOLUP
c       DOUBLE PRECISION XWGTUP,SCALUP,AQEDUP,AQCDUP,PUP,VTIMUP,SPINUP
c       COMMON/HEPEUP/NUP,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP,IDUP(MAXNUP),
c      &ISTUP(MAXNUP),MOTHUP(2,MAXNUP),ICOLUP(2,MAXNUP),PUP(5,MAXNUP),
c      &VTIMUP(MAXNUP),SPINUP(MAXNUP)
c       SAVE /HEPEUP/

C...Read info from file.
c       IF(MSTP(162).GT.0) THEN
c         READ(MSTP(162),*,END=110,ERR=110) NUP,IDPRUP,XWGTUP,SCALUP,
c      &  AQEDUP,AQCDUP
c         DO 100 I=1,NUP
c           READ(MSTP(162),*,END=110,ERR=110) IDUP(I),ISTUP(I),
c      &    MOTHUP(1,I),MOTHUP(2,I),ICOLUP(1,I),ICOLUP(2,I),
c      &    (PUP(J,I),J=1,5),VTIMUP(I),SPINUP(I)
c 100     CONTINUE
c         RETURN
C...Special when reached end of file or other error.
c 110     NUP=0

C...Else not implemented.
c       ELSE
c         WRITE(MSTU(11),5000)
c         STOP
c       ENDIF

C...Format for error printout.
c 5000 FORMAT(1X,'Error: You have not implemented UPEVNT routine'/
c      &1X,'Dummy routine in PYTHIA file called instead.'/
c      &1X,'Execution stopped!')

c       RETURN
c       END

C*******************************************************************

C...UPVETO
C...Dummy routine, to be replaced by user, to veto event generation
C...on the parton level, after parton showers but before multiple
C...interactions, beam remnants and hadronization is added.
C...If resonances like W, Z, top, Higgs and SUSY particles are handed
C...undecayed from UPEVNT, or are generated by PYTHIA, they will also
C...be undecayed at this stage; if decayed their decay products will
C...have been allowed to shower.

C...All partons at the end of the shower phase are stored in the
```

2263

```
C...HEPEVT commonblock. The interesting information is
C...NHEP = the number of such partons, in entries 1 <= i <= NHEP,
C...IDHEP(I) = the particle ID code according to PDG conventions,
C...PHEP(J,I) = the (p_x, p_y, p_z, E, m) of the particle.
C...All ISTHEP entries are 1, while the rest is zeroed.

C...The user decision is to be conveyed by the IVETO value.
C...IVETO = 0 : retain current event and generate in full;
C...      = 1 : abort generation of current event and move to next.

      SUBROUTINE UPVETO(IVETO)

C...HEPEVT commonblock.
      PARAMETER (NMXHEP=4000)
      COMMON/HEPEVT/NEVHEP,NHEP,ISTHEP(NMXHEP),IDHEP(NMXHEP),
     &JMOHEP(2,NMXHEP),JDAHEP(2,NMXHEP),PHEP(5,NMXHEP),VHEP(4,NMXHEP)
      DOUBLE PRECISION PHEP,VHEP
      SAVE /HEPEVT/

C...Next few lines allow you to see what info PYVETO extracted from
C...the full event record for the first two events.
C...Delete if you don't want it.
      DATA NLIST/0/
      SAVE NLIST
      IF(NLIST.LE.2) THEN
        WRITE(*,*) ' Full event record at time of UPVETO call:'
        CALL PYLIST(1)
        WRITE(*,*) ' Part of event record made available to UPVETO:'
        CALL PYLIST(5)
        NLIST=NLIST+1
      ENDIF

C...Make decision here.
      IVETO = 0

      RETURN
      END
```

16.5 Matching

16.5.1 MLM matching

⟨mlm_matching.f90⟩≡
  ⟨*File header*⟩

  module mlm_matching

  ⟨*Use kinds*⟩
    use kinds, only: double !NODEP!
    use constants !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!

```
    use file_utils !NODEP!
```

⟨*Standard module head*⟩

⟨*MLM matching: public*⟩

⟨*MLM matching: types*⟩

```
  contains
```

⟨*MLM matching: procedures*⟩

```
  end module mlm_matching
```

This is a container for the (colored) parton momenta as well as the jet momenta.

⟨*MLM matching: public*⟩≡
```
    public :: mlm_matching_data_t
```

⟨*MLM matching: types*⟩≡
```
    type :: mlm_matching_data_t
       logical :: is_hadron_collision = .false.
       type(vector4_t), dimension(:), allocatable, public :: P_ME
       type(vector4_t), dimension(:), allocatable, public :: P_PS
       type(vector4_t), dimension(:), allocatable, private :: JETS_ME
       type(vector4_t), dimension(:), allocatable, private :: JETS_PS
    end type mlm_matching_data_t
```

⟨*MLM matching: public*⟩+≡
```
    public :: mlm_matching_settings_t
```

⟨*MLM matching: types*⟩+≡
```
    type :: mlm_matching_settings_t
       real(default) :: mlm_Qcut_ME = one
       real(default) :: mlm_Qcut_PS = one
       real(default) :: mlm_ptmin, mlm_etamax, mlm_Rmin, mlm_Emin
       real(default) :: mlm_ETclusfactor = 0.2_default
       real(default) :: mlm_ETclusminE = five
       real(default) :: mlm_etaclusfactor = one
       real(default) :: mlm_Rclusfactor = one
       real(default) :: mlm_Eclusfactor = one
       integer :: kt_imode_hadronic = 4313
       integer :: kt_imode_leptonic = 1111
       integer :: mlm_nmaxMEjets = 0
    end type mlm_matching_settings_t
```

⟨*MLM matching: public*⟩+≡
```
    public :: mlm_matching_settings_write
```

⟨*MLM matching: procedures*⟩≡
```
    subroutine mlm_matching_settings_write (settings, unit)
      type(mlm_matching_settings_t), intent(in) :: settings
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(3x,A,ES19.12)") &
```

```
              "mlm_Qcut_ME              = ", settings%mlm_Qcut_ME
      write (u, "(3x,A,ES19.12)") &
              "mlm_Qcut_PS              = ", settings%mlm_Qcut_PS
      write (u, "(3x,A,ES19.12)") &
              "mlm_ptmin                = ", settings%mlm_ptmin
      write (u, "(3x,A,ES19.12)") &
              "mlm_etamax               = ", settings%mlm_etamax
      write (u, "(3x,A,ES19.12)") &
              "mlm_Rmin                 = ", settings%mlm_Rmin
      write (u, "(3x,A,ES19.12)") &
              "mlm_Emin                 = ", settings%mlm_Emin
      write (u, "(3x,A,1x,I0)") &
              "mlm_nmaxMEjets           = ", settings%mlm_nmaxMEjets
      write (u, "(3x,A,ES19.12)") &
              "mlm_ETclusfactor  (D=0.2)   = ", settings%mlm_ETclusfactor
      write (u, "(3x,A,ES19.12)") &
              "mlm_ETclusminE    (D=5.0)   = ", settings%mlm_ETclusminE
      write (u, "(3x,A,ES19.12)") &
              "mlm_etaclusfactor (D=1.0)   = ", settings%mlm_etaClusfactor
      write (u, "(3x,A,ES19.12)") &
              "mlm_Rclusfactor   (D=1.0)   = ", settings%mlm_RClusfactor
      write (u, "(3x,A,ES19.12)") &
              "mlm_Eclusfactor   (D=1.0)   = ", settings%mlm_EClusfactor
    end subroutine mlm_matching_settings_write
```

⟨*MLM matching: public*⟩+≡
```
    public :: mlm_matching_data_write
```

⟨*MLM matching: procedures*⟩+≡
```
    subroutine mlm_matching_data_write (data, unit)
      type(mlm_matching_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      integer :: i
      integer :: u
      u = output_unit (unit);  if (u < 0)  return

      write (u, "(1x,A)") "MLM matching data:"
      write (u, "(3x,A)") "Momenta of ME partons:"
      if (allocated (data%P_ME)) then
         do i = 1, size (data%P_ME)
            write (u, "(4x)", advance = "no")
            call vector4_write (data%P_ME(i), unit = u)
         end do
      else
         write (u, "(5x,A)")  "[empty]"
      end if
      call write_separator (u)
      write (u, "(3x,A)")  "Momenta of ME jets:"
      if (allocated (data%JETS_ME)) then
         do i = 1, size (data%JETS_ME)
            write (u, "(4x)", advance = "no")
            call vector4_write (data%JETS_ME(i), unit = u)
         end do
      else
```

2266

```
          write (u, "(5x,A)")  "[empty]"
       end if
       call write_separator (u)
       write(u, "(3x,A)")  "Momenta of shower partons:"
       if (allocated (data%P_PS)) then
          do i = 1, size (data%P_PS)
             write (u, "(4x)", advance = "no")
             call vector4_write (data%P_PS(i), unit = u)
          end do
       else
          write (u, "(5x,A)")  "[empty]"
       end if
       call write_separator (u)
       write (u, "(3x,A)")  "Momenta of shower jets:"
       if (allocated (data%JETS_PS)) then
          do i = 1, size (data%JETS_PS)
             write (u, "(4x)", advance = "no")
             call vector4_write (data%JETS_PS(i), unit = u)
          end do
       else
          write (u, "(5x,A)")  "[empty]"
       end if
       call write_separator (u)
    end subroutine mlm_matching_data_write
```

⟨*MLM matching: public*⟩+≡
```
    public :: mlm_matching_data_final
```

⟨*MLM matching: procedures*⟩+≡
```
    subroutine mlm_matching_data_final (data)
      type(mlm_matching_data_t), intent(inout) :: data
      ! if (allocated (data%P_ME))  deallocate(data%P_ME)
      if (allocated (data%P_PS))  deallocate (data%P_PS)
      if (allocated (data%JETS_ME))  deallocate(data%JETS_ME)
      if (allocated (data%JETS_PS))  deallocate(data%JETS_PS)
    end subroutine mlm_matching_data_final
```

⟨*MLM matching: public*⟩+≡
```
    public :: mlm_matching_apply
```

⟨*MLM matching: procedures*⟩+≡
```
    subroutine mlm_matching_apply (data, settings, vetoed)
      type(mlm_matching_data_t), intent(inout) :: data
      type(mlm_matching_settings_t), intent(in) :: settings
      logical, intent(out) :: vetoed

      integer :: i, j
      integer :: n_jets_ME, n_jets_PS, n_jets_PS_atycut
      real(double) :: ycut
      real(double), dimension(:, :), allocatable :: PP
      real(double), dimension(:), allocatable :: Y
      real(double), dimension(:,:), allocatable :: P_JETS
      real(double), dimension(:,:), allocatable :: P_ME
      integer, dimension(:), allocatable :: JET
```

```
      integer :: NJET, NSUB
      integer :: imode
!!! TODO: (bcn 2014-03-26) Why is ECUT hard coded to 1?
!!! It is the denominator of the KT measure. Candidate for removal
      real(double) :: ECUT = 1._double
      integer :: ip1,ip2

      ! KTCLUS COMMON BLOCK
      INTEGER NMAX,NUM,HIST
      PARAMETER (NMAX=512)
      DOUBLE PRECISION P,KT,KTP,KTS,ETOT,RSQ,KTLAST
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM

      vetoed = .true.
      if (signal_is_pending ())  return
```

⟨*Set* `n_jets_ME/PS` *from* `data` *(or equal zero)*⟩

⟨*Jet clustering for partons after matrix element*⟩

⟨*Jet clustering for partons after shower*⟩

```
      ! call data_write(data)
```

⟨*Veto: too many or not enough jets after PS*⟩

⟨*Cluster ME jets with PS jets one at a time*⟩

```
      vetoed = .false.
  999 continue
    end subroutine mlm_matching_apply
```

⟨*Set* `n_jets_ME/PS` *from* `data` *(or equal zero)*⟩≡
```
  if (allocated (data%P_ME)) then
     ! print *, "number of partons after ME: ", size(data%P_ME)
     n_jets_ME = size (data%P_ME)
  else
     n_jets_ME = 0
  end if
  if (allocated (data%p_PS)) then
     ! print *, "number of partons after PS: ", size(data%p_PS)
     n_jets_PS = size (data%p_PS)
  else
     n_jets_PS = 0
  end if
```

⟨*Jet clustering for partons after matrix element*⟩≡
```
  if (n_jets_ME > 0) then
     ycut = (settings%mlm_ptmin)**2
     ! ycut = settings%mlm_Qcut_ME**2
     allocate (PP(1:4, 1:N_jets_ME))
     do i = 1, n_jets_ME
        PP(1,i) = vector4_get_component (data%p_ME(i), 1)
```

```
         PP(2,i) = vector4_get_component (data%p_ME(i), 2)
         PP(3,i) = vector4_get_component (data%p_ME(i), 3)
         PP(4,i) = vector4_get_component (data%p_ME(i), 0)
      end do

      ⟨Set imode for lepton or hadron collisions⟩

      allocate (P_ME(1:4,1:n_jets_ME))
      allocate (JET(1:n_jets_ME))
      allocate (Y(1:n_jets_ME))

   !!! TODO: (bcn 2014-03-26) has he forgotten mlm_Rclusfactor here? #
      if (signal_is_pending ())  return
      call KTCLUR (imode, PP, n_jets_ME, &
           dble (settings%mlm_Rclusfactor * settings%mlm_Rmin), ECUT, y, *999)
      call KTRECO (1, PP, n_jets_ME, ECUT, ycut, ycut, P_ME, JET, &
           NJET, NSUB, *999)

      n_jets_ME = NJET
      if (NJET > 0) then
         allocate (data%JETS_ME (1:NJET))
         do i = 1, NJET
            data%JETS_ME(i) = vector4_moving (REAL(P_ME(4,i), default), &
                 vector3_moving([REAL(P_ME(1,i), default), &
                 REAL(P_ME(2,i), default), REAL(P_ME(3,i), default)]))
         end do
      end if
      deallocate (P_ME)
      deallocate (JET)
      deallocate (Y)
      deallocate (PP)
   end if
⟨Jet clustering for partons after shower⟩≡
   if (n_jets_PS > 0) then
      ycut = (settings%mlm_ptmin + max (settings%mlm_ETclusminE, &
           settings%mlm_ETclusfactor * settings%mlm_ptmin))**2
      ! ycut = settings%mlm_Qcut_PS**2
      allocate (PP(1:4, 1:n_jets_PS))
      do i = 1, n_jets_PS
         PP(1,i) = vector4_get_component (data%p_PS(i), 1)
         PP(2,i) = vector4_get_component (data%p_PS(i), 2)
         PP(3,i) = vector4_get_component (data%p_PS(i), 3)
         PP(4,i) = vector4_get_component (data%p_PS(i), 0)
      end do

      ⟨Set imode for lepton or hadron collisions⟩

      allocate (P_JETS(1:4,1:n_jets_PS))
      allocate (JET(1:n_jets_PS))
      allocate (Y(1:n_jets_PS))

      if (signal_is_pending ()) return
      call KTCLUR (imode, PP, n_jets_PS, &
           dble (settings%mlm_Rclusfactor * settings%mlm_Rmin), &
```

```
          ECUT, y, *999)
      call KTRECO (1, PP, n_jets_PS, ECUT, ycut, ycut, P_JETS, JET, &
          NJET, NSUB, *999)
      n_jets_PS_atycut = NJET
      if (n_jets_ME == settings%mlm_nmaxMEjets .and. NJET > 0) then
          ! print *, " resetting ycut to ", Y(settings%mlm_nmaxMEjets)
          ycut = y(settings%mlm_nmaxMEjets)
          call KTRECO (1, PP, n_jets_PS, ECUT, ycut, ycut, P_JETS, JET, &
              NJET, NSUB, *999)
      end if

      ! !Sample of code for a FastJet interface
      ! palg = 1d0          ! 1.0d0 = kt, 0.0d0 = Cam/Aachen, -1.0d0 = anti-kt
      ! R = 0.7_double      ! radius parameter
      ! f = 0.75_double     ! overlap threshold
      ! !call fastjetppgenkt(PP,n,R,palg,P_JETS,NJET)    ! KT-Algorithm
      ! !call fastjetsiscone(PP,n,R,f,P_JETS,NJET)       ! SiSCone-Algorithm

      if (NJET > 0) then
          allocate (data%JETS_PS(1:NJET))
          do i = 1, NJET
              data%JETS_PS(i) = vector4_moving (REAL(P_JETS(4,i), default), &
                  vector3_moving([REAL(P_JETS(1,i), default), &
                  REAL(P_JETS(2,i), default), REAL(P_JETS(3,i), default)]))
          end do
      end if

      deallocate (P_JETS)
      deallocate (JET)
      deallocate (Y)
  else
      n_jets_PS_atycut = 0
  end if
⟨Set imode for lepton or hadron collisions⟩≡
  if (data%is_hadron_collision) then
      imode = settings%kt_imode_hadronic
  else
      imode = settings%kt_imode_leptonic
  end if
⟨Veto: too many or not enough jets after PS⟩≡
  if (n_jets_PS_atycut < n_jets_ME) then
      ! print *, "DISCARDING: Not enough PS jets: ", n_jets_PS_atycut
      return
  end if
  if (n_jets_PS_atycut > n_jets_ME .and. n_jets_ME /= settings%mlm_nmaxMEjets) then
      ! print *, "DISCARDING: Too many PS jets: ", n_jets_PS_atycut
      return
  end if
⟨Cluster ME jets with PS jets one at a time⟩≡
  if (allocated(data%JETS_PS)) then
      ! print *, "number of jets after PS: ", size(data%JETS_PS)
      n_jets_PS = size (data%JETS_PS)
  else
```

```
      n_jets_PS = 0
end if
if (n_jets_ME > 0 .and. n_jets_PS > 0) then
   n_jets_PS = size (data%JETS_PS)
   if (allocated (PP))  deallocate(PP)
   allocate (PP(1:4, 1:n_jets_PS + 1))
   do i = 1, n_jets_PS
      if (signal_is_pending ()) return
      ! call vector4_write(data%jets_PS(i))
      PP(1,i) = vector4_get_component (data%JETS_PS(i), 1)
      PP(2,i) = vector4_get_component (data%JETS_PS(i), 2)
      PP(3,i) = vector4_get_component (data%JETS_PS(i), 3)
      PP(4,i) = vector4_get_component (data%JETS_PS(i), 0)
   end do
   if (allocated (Y))  deallocate(Y)
   allocate (Y(1:n_jets_PS + 1))
   y = zero
   do i = 1, n_jets_ME
      ! print *, "ME JET"
      ! call vector4_write(data%jets_ME(i))
      PP(1,n_jets_PS + 2 - i) = vector4_get_component (data%JETS_ME(i), 1)
      PP(2,n_jets_PS + 2 - i) = vector4_get_component (data%JETS_ME(i), 2)
      PP(3,n_jets_PS + 2 - i) = vector4_get_component (data%JETS_ME(i), 3)
      PP(4,n_jets_PS + 2 - i) = vector4_get_component (data%JETS_ME(i), 0)
      !!! This makes more sense than hardcoding
      ! call KTCLUS (4313, PP, (n_jets_PS + 2 - i), 1.0_double, Y, *999)
      call KTCLUR (imode, PP, (n_jets_PS + 2 - i), &
        dble (settings%mlm_Rclusfactor * settings%mlm_Rmin), &
        ECUT, y, *999)
      ! print *, "    Y=" , y
      ! print *, y(n_jets_PS + 1 - i), " " , ycut
      if (0.99 * y(n_jets_PS + 1 - (i - 1)).gt.ycut) then
         ! print *, "DISCARDING: Jet ", i, " not clusterd"
          return
      end if
      !!! search for and remove PS jet clustered with ME Jet
      ! print *, "i=",  i, n_jets_PS, n_jets_ME
      ! do j = 1, n_jets_PS + 2 - i
      !    print *, PP(1,j), PP(2,j), PP(3,j), PP(4,j)
      ! end do
      ! print *, " n_jets_PS=", n_jets_PS
      ip1 = HIST(n_jets_PS + 2 - i) / NMAX
      ip2 = mod(hist(n_jets_PS + 2 - i), NMAX)
      if ((ip2 /= n_jets_PS + 2 - i) .or. (ip1 <= 0)) then
         ! print *, "DISCARDING: Jet ", i, " not clustered ", ip1, ip2, &
         !      hist(n_jets_PS + 2 - i)
         return
      else
         ! print *, "PARTON clustered", ip1, ip2, hist(n_jets_PS + 2 - i)
         PP(:,IP1) = zero
         do j = IP1, n_jets_PS - i
            PP(:, j) = PP(:,j + 1)
         end do
      end if
```

2271

```
          end do
      end if
```

⟨*MLM matching: procedures*⟩+≡

```
  !C----------------------------------------------------------------------
  !C----------------------------------------------------------------------
  !C----------------------------------------------------------------------
  !C      KTCLUS: written by Mike Seymour, July 1992.
  !C      Last modified November 2000.
  !C      Please send comments or suggestions to Mike.Seymour@rl.ac.uk
  !C
  !C      This is a general-purpose kt clustering package.
  !C      It can handle ee, ep and pp collisions.
  !C      It is loosely based on the program of Siggi Bethke.
  !C
  !C      The time taken (on a 10MIP machine) is (0.2microsec)*N**3
  !C      where N is the number of particles.
  !C      Over 90 percent of this time is used in subroutine KTPMIN, which
  !C      simply finds the minimum member of a one-dimensional array.
  !C      It is well worth thinking about optimization: on the SPARCstation
  !C      a factor of two increase was obtained simply by increasing the
  !C      optimization level from its default value.
  !C
  !C      The approach is to separate the different stages of analysis.
  !C      KTCLUS does all the clustering and records a merging history.
  !C      It returns a simple list of the y values at which each merging
  !C      occured. Then the following routines can be called to give extra
  !C      information on the most recently analysed event.
  !C      KTCLUR is identical but includes an R parameter, see below.
  !C      KTYCUT gives the number of jets at each given YCUT value.
  !C      KTYSUB gives the number of sub-jets at each given YCUT value.
  !C      KTBEAM gives same info as KTCLUS but only for merges with the beam
  !C      KTJOIN gives same info as KTCLUS but for merges of sub-jets.
  !C      KTRECO reconstructs the jet momenta at a given value of YCUT.
  !C      It also gives information on which jets at scale YCUT belong to
  !C      which macro-jets at scale YMAC, for studying sub-jet properties.
  !C      KTINCL reconstructs the jet momenta according to the inclusive jet
  !C      definition of Ellis and Soper.
  !C      KTISUB, KTIJOI and KTIREC are like KTYSUB, KTJOIN and KTRECO,
  !C      except that they only apply to one inclusive jet at a time,
  !C      with the pt of that jet automatically used for ECUT.
  !C      KTWICH gives a list of which particles ended up in which jets.
  !C      KTWCHS gives the same thing, but only for subjets.
  !C      Note that the numbering of jets used by these two routines is
  !C      guaranteed to be the same as that used by KTRECO.
  !C
  !C      The collision type and analysis type are indicated by the first
  !C      argument of KTCLUS. IMODE=<TYPE><ANGLE><MONO><RECOM> where
  !C      TYPE:  1=>ee, 2=>ep with p in -z direction, 3=>pe, 4=>pp
  !C      ANGLE: 1=>angular kt def., 2=>DeltaR, 3=>f(DeltaEta,DeltaPhi)
  !C             where f()=2(cosh(eta)-cos(phi)) is the QCD emission metric
  !C      MONO:  1=>derive relative pseudoparticle angles from jets
  !C             2=>monotonic definitions of relative angles
  !C      RECOM: 1=>E recombination scheme, 2=>pt scheme, 3=>pt**2 scheme
  !C
```

```
!C     There are also abbreviated forms for the most common combinations:
!C     IMODE=1 => E scheme in e+e-                        (=1111)
!C          2 => E scheme in ep                           (=2111)
!C          3 => E scheme in pe                           (=3111)
!C          4 => E scheme in pp                           (=4111)
!C          5 => covariant E scheme in pp                 (=4211)
!C          6 => covariant pt-scheme in pp                (=4212)
!C          7 => covariant monotonic pt**2-scheme in pp   (=4223)
!C
!C     KTRECO no longer needs to reconstruct the momenta according to the
!C     same recombination scheme in which they were clustered. Its first
!C     argument gives the scheme, taking the same values as RECOM above.
!C
!C     Note that unlike previous versions, all variables which hold y
!C     values have been named in a consistent way:
!C     Y()  is the output scale at which jets were merged,
!C     YCUT is the input scale at which jets should be counted, and
!C          jet-momenta reconstructed etc,
!C     YMAC is the input macro-jet scale, used in determining whether
!C          or not each jet is a sub-jet.
!C     The original scheme defined in our papers is equivalent to always
!C     setting YMAC=1.
!C     Whenever a YCUT or YMAC variable is used, it is rounded down
!C     infinitesimally, so that for example, setting YCUT=Y(2) refers
!C     to the scale where the event is 2-jet, even if rounding errors
!C     have shifted its value slightly.
!C
!C     An R parameter can be used in hadron-hadron collisions by
!C     calling KTCLUR instead of KTCLUS.  This is as suggested by
!C     Ellis and Soper, but implemented slightly differently,
!C     as in M.H. Seymour, LU TP 94/2 (submitted to Nucl. Phys. B.).
!C     R**2 multiplies the single Kt everywhere it is used.
!C     Calling KTCLUR with R=1 is identical to calling KTCLUS.
!C     R plays a similar role to the jet radius in a cone-type algorithm,
!C     but is scaled up by about 40% (ie R=0.7 in a cone algorithm is
!C     similar to this algorithm with R=1).
!C     Note that R.EQ.1 must be used for the e+e- and ep versions,
!C     and is strongly recommended for the hadron-hadron version.
!C     However, R values smaller than 1 have been found to be useful for
!C     certain applications, particularly the mass reconstruction of
!C     highly-boosted colour-singlets such as high-pt hadronic Ws,
!C     as in M.H. Seymour, LU TP 93/8 (to appear in Z. Phys. C.).
!C     Situations in which R<1 is useful are likely to also be those in
!C     which the inclusive reconstruction method is more useful.
!C
!C     Also included is a set of routines for doing Lorentz boosts:
!C     KTLBST finds the boost matrix to/from the cm frame of a 4-vector
!C     KTRROT finds the rotation matrix from one vector to another
!C     KTMMUL multiplies together two matrices
!C     KTVMUL multiplies a vector by a matrix
!C     KTINVT inverts a transformation matrix (nb NOT a general 4 by 4)
!C     KTFRAM boosts a list of vectors between two arbitrary frames
!C     KTBREI boosts a list of vectors between the lab and Breit frames
!C     KTHADR boosts a list of vectors between the lab and hadronic cmf
```

```
!C        The last two need the momenta in the +z direction of the lepton
!C        and hadron beams, and the 4-momentum of the outgoing lepton.
!C
!C    The main reference is:
!C       S. Catani, Yu.L. Dokshitzer, M.H. Seymour and B.R. Webber,
!C          Nucl.Phys.B406(1993)187.
!C    The ep version was proposed in:
!C       S. Catani, Yu.L. Dokshitzer and B.R. Webber,
!C          Phys.Lett.285B(1992)291.
!C    The inclusive reconstruction method was proposed in:
!C       S.D. Ellis and D.E. Soper,
!C          Phys.Rev.D48(1993)3160.
!C
!C----------------------------------------------------------------------
!C----------------------------------------------------------------------
!C----------------------------------------------------------------------
  SUBROUTINE KTCLUS(IMODE,PP,NN,ECUT,Y,*)
    IMPLICIT NONE
!C---DO CLUSTER ANALYSIS OF PARTICLES IN PP
!C
!C   IMODE  = INPUT  : DESCRIBED ABOVE
!C   PP(I,J) = INPUT  : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C   NN      = INPUT  : NUMBER OF PARTICLES
!C   ECUT    = INPUT  : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   Y(J)    = OUTPUT : VALUE OF Y FOR WHICH EVENT CHANGES FROM BEING
!C                        J JET TO J-1 JET
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED (MOST LIKELY DUE TO TOO MANY PARTICLES)
!C
!C   NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C   AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
    INTEGER IMODE,NN
    DOUBLE PRECISION PP(4,*)
    DOUBLE PRECISION ECUT,Y(*),ONE
    ONE=1
    CALL KTCLUR(IMODE,PP,NN,ONE,ECUT,Y,*999)
    RETURN
999 RETURN 1
  END SUBROUTINE KTCLUS
!C----------------------------------------------------------------------
  SUBROUTINE KTCLUR(IMODE,PP,NN,R,ECUT,Y,*)
    IMPLICIT NONE
!C---DO CLUSTER ANALYSIS OF PARTICLES IN PP
!C
!C   IMODE  = INPUT  : DESCRIBED ABOVE
!C   PP(I,J) = INPUT  : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C   NN      = INPUT  : NUMBER OF PARTICLES
!C   R       = INPUT  : ELLIS AND SOPER'S R PARAMETER, SEE ABOVE.
!C   ECUT    = INPUT  : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   Y(J)    = OUTPUT : VALUE OF Y FOR WHICH EVENT CHANGES FROM BEING
!C                        J JET TO J-1 JET
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED (MOST LIKELY DUE TO TOO MANY PARTICLES)
```

2274

```
!C
!C   NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C   AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,IM,IMODE,TYPE,ANGL,MONO,RECO,N,I,J,NN, &
          IMIN,JMIN,KMIN,NUM,HIST,INJET,IABBR,NABBR
      PARAMETER (NMAX=512,NABBR=7)
      DOUBLE PRECISION PP(4,*)
      integer :: u
!CHANGE    DOUBLE PRECISION R,ECUT,Y(*),P,KT,ETOT,RSQ,KTP,KTS,KTPAIR,KTSING, &
      DOUBLE PRECISION R,ECUT,Y(*),P,KT,ETOT,RSQ,KTP,KTS, &
          KTMIN,ETSQ,KTLAST,KTMAX,KTTMP
      LOGICAL FIRST
      CHARACTER TITLE(4,4)*10
!C---KT RECORDS THE KT**2 OF EACH MERGING.
!C---KTLAST RECORDS FOR EACH MERGING, THE HIGHEST ECUT**2 FOR WHICH THE
!C   RESULT IS NOT MERGED WITH THE BEAM (COULD BE LARGER THAN THE
!C   KT**2 AT WHICH IT WAS MERGED IF THE KT VALUES ARE NOT MONOTONIC).
!C   THIS MAY SOUND POINTLESS, BUT ITS USEFUL FOR DETERMINING WHETHER
!C   SUB-JETS SURVIVED TO SCALE Y=YMAC OR NOT.
!C---HIST RECORDS MERGING HISTORY:
!C   N=>DELETED TRACK N, M*NMAX+N=>MERGED TRACKS M AND N (M<N).
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
      DIMENSION INJET(NMAX),IABBR(NABBR)
      DATA FIRST,TITLE,IABBR/.TRUE., &
          'e+e-      ','ep        ','pe        ','pp        ', &
          'angle     ','DeltaR    ','f(DeltaR) ','********** ', &
          'no        ','yes       ','********** ','********** ', &
          'E         ','Pt        ','Pt**2     ','********** ', &
          1111,2111,3111,4111,4211,4212,4223/
!C---CHECK INPUT
      IM=IMODE
      IF (IM.GE.1.AND.IM.LE.NABBR) IM=IABBR(IM)
      TYPE=MOD(IM/1000,10)
      ANGL=MOD(IM/100 ,10)
      MONO=MOD(IM/10  ,10)
      RECO=MOD(IM     ,10)
      IF (NN.GT.NMAX) CALL KTWARN('KT-MAX',100,*999)
      IF (NN.LT.1) CALL KTWARN('KT-LT1',100,*999)
      IF (NN.LT.2.AND.TYPE.EQ.1) CALL KTWARN('KT-LT2',100,*999)
      IF (TYPE.LT.1.OR.TYPE.GT.4.OR.ANGL.LT.1.OR.ANGL.GT.4.OR. &
          MONO.LT.1.OR.MONO.GT.2.OR.RECO.LT.1.OR.RECO.GT.3) CALL KTWARN('KTCLUS',101,*999)
      u = output_unit ()
      IF (FIRST) THEN
         WRITE (u,'(/,1X,54(''*'')/A)') &
             ' KTCLUS: written by Mike Seymour, July 1992.'
         WRITE (u,'(A)') &
             ' Last modified November 2000.'
         WRITE (u,'(A)') &
             ' Please send comments or suggestions to Mike.Seymour@rl.ac.uk'
         WRITE (u,'(/A,I2,2A)') &
             '          Collision type =',TYPE,' = ',TITLE(TYPE,1)
         WRITE (u,'(A,I2,2A)') &
```

```
                  '      Angular variable =',ANGL,' = ',TITLE(ANGL,2)
            WRITE (u,'(A,I2,2A)') &
                  ' Monotonic definition =',MONO,' = ',TITLE(MONO,3)
            WRITE (u,'(A,I2,2A)') &
                  ' Recombination scheme =',RECO,' = ',TITLE(RECO,4)
            IF (R.NE.1) THEN
               WRITE (u,'(A,F5.2)') &
                     '      Radius parameter =',R
               IF (TYPE.NE.4) WRITE (u,'(A)') &
                     ' R.NE.1 is strongly discouraged for this collision type!'
            ENDIF
            WRITE (u,'(1X,54(''*'')/)')
            FIRST=.FALSE.
         ENDIF
!C---COPY PP TO P
      N=NN
      NUM=NN
      CALL KTCOPY(PP,N,P,(RECO.NE.1))
      ETOT=0
      DO I=1,N
         ETOT=ETOT+P(4,I)
      END DO
      IF (ETOT.EQ.0) CALL KTWARN('KTCLUS',102,*999)
      IF (ECUT.EQ.0) THEN
         ETSQ=1/ETOT**2
      ELSE
         ETSQ=1/ECUT**2
      ENDIF
      RSQ=R**2
!C---CALCULATE ALL PAIR KT's
      DO I=1,N-1
         DO J=I+1,N
            KTP(J,I)=-1
            KTP(I,J)=KTPAIR(ANGL,P(1,I),P(1,J),KTP(J,I))
         END DO
      END DO
!C---CALCULATE ALL SINGLE KT's
      DO I=1,N
         KTS(I)=KTSING(ANGL,TYPE,P(1,I))
      END DO
      KTMAX=0
!C---MAIN LOOP
300   CONTINUE
!C---FIND MINIMUM MEMBER OF KTP
      CALL KTPMIN(KTP,NMAX,N,IMIN,JMIN)
!C---FIND MINIMUM MEMBER OF KTS
      CALL KTSMIN(KTS,NMAX,N,KMIN)
!C---STORE Y VALUE OF TRANSITION FROM N TO N-1 JETS
      KTMIN=KTP(IMIN,JMIN)
      KTTMP=RSQ*KTS(KMIN)
      IF ((TYPE.GE.2.AND.TYPE.LE.4).AND. &
          (KTTMP.LE.KTMIN.OR.N.EQ.1)) &
            KTMIN=KTTMP
      KT(N)=KTMIN
```

2276

```
      Y(N)=KT(N)*ETSQ
!C---IF MONO.GT.1, SEQUENCE IS SUPPOSED TO BE MONOTONIC, IF NOT, WARN
      IF (KTMIN.LT.KTMAX.AND.MONO.GT.1) CALL KTWARN('KTCLUS',1,*999)
      IF (KTMIN.GE.KTMAX) KTMAX=KTMIN
!C---IF LOWEST KT IS TO A BEAM, THROW IT AWAY AND MOVE LAST ENTRY UP
      IF (KTMIN.EQ.KTTMP) THEN
         CALL KTMOVE(P,KTP,KTS,NMAX,N,KMIN,1)
!C---UPDATE HISTORY AND CROSS-REFERENCES
         HIST(N)=KMIN
         INJET(N)=KMIN
         DO I=N,NN
            IF (INJET(I).EQ.KMIN) THEN
               KTLAST(I)=KTMAX
               INJET(I)=0
            ELSEIF (INJET(I).EQ.N) THEN
               INJET(I)=KMIN
            ENDIF
         END DO
!C---OTHERWISE MERGE JETS IMIN AND JMIN AND MOVE LAST ENTRY UP
      ELSE
         CALL KTMERG(P,KTP,KTS,NMAX,IMIN,JMIN,N,TYPE,ANGL,MONO,RECO)
         CALL KTMOVE(P,KTP,KTS,NMAX,N,JMIN,1)
!C---UPDATE HISTORY AND CROSS-REFERENCES
         HIST(N)=IMIN*NMAX+JMIN
         INJET(N)=IMIN
         DO I=N,NN
            IF (INJET(I).EQ.JMIN) THEN
               INJET(I)=IMIN
            ELSEIF (INJET(I).EQ.N) THEN
               INJET(I)=JMIN
            ENDIF
         END DO
      ENDIF
!C---THATS ALL THERE IS TO IT
      N=N-1
      IF (N.GT.1 .OR. N.GT.0.AND.(TYPE.GE.2.AND.TYPE.LE.4)) GOTO 300
      IF (N.EQ.1) THEN
         KT(N)=1D20
         Y(N)=KT(N)*ETSQ
      ENDIF
      RETURN
999   RETURN 1
  END SUBROUTINE KTCLUR
!C-------------------------------------------------------------------
      SUBROUTINE KTYCUT(ECUT,NY,YCUT,NJET,*)
      IMPLICIT NONE
!C---COUNT THE NUMBER OF JETS AT EACH VALUE OF YCUT, FOR EVENT WHICH HAS
!C   ALREADY BEEN ANALYSED BY KTCLUS.
!C
!C   ECUT   = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   NY     = INPUT : NUMBER OF YCUT VALUES
!C   YCUT(J) = INPUT : Y VALUES AT WHICH NUMBERS OF JETS ARE COUNTED
!C   NJET(J) =OUTPUT : NUMBER OF JETS AT YCUT(J)
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
```

```
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NY,NJET(NY),NMAX,HIST,I,J,NUM
      PARAMETER (NMAX=512)
      DOUBLE PRECISION YCUT(NY),ETOT,RSQ,P,KT,KTP,KTS,ETSQ,ECUT,KTLAST, &
          ROUND
      PARAMETER (ROUND=0.99999D0)
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
      IF (ETOT.EQ.0) CALL KTWARN('KTYCUT',100,*999)
      IF (ECUT.EQ.0) THEN
         ETSQ=1/ETOT**2
      ELSE
         ETSQ=1/ECUT**2
      ENDIF
      DO I=1,NY
         NJET(I)=0
      END DO
      DO I=NUM,1,-1
         DO J=1,NY
            IF (NJET(J).EQ.0.AND.KT(I)*ETSQ.GE.ROUND*YCUT(J)) NJET(J)=I
         END DO
      END DO
      RETURN
999   RETURN 1
      END SUBROUTINE KTYCUT
!C---------------------------------------------------------------------
      SUBROUTINE KTYSUB(ECUT,NY,YCUT,YMAC,NSUB,*)
       IMPLICIT NONE
!C---COUNT THE NUMBER OF SUB-JETS AT EACH VALUE OF YCUT, FOR EVENT WHICH
!C   HAS ALREADY BEEN ANALYSED BY KTCLUS.
!C   REMEMBER THAT A SUB-JET IS DEFINED AS A JET AT Y=YCUT WHICH HAS NOT
!C   YET BEEN MERGED WITH THE BEAM AT Y=YMAC.
!C
!C   ECUT    = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   NY      = INPUT : NUMBER OF YCUT VALUES
!C   YCUT(J) = INPUT : Y VALUES AT WHICH NUMBERS OF SUB-JETS ARE COUNTED
!C   YMAC    = INPUT : Y VALUE USED TO DEFINE MACRO-JETS, TO DETERMINE
!C                         WHICH JETS ARE SUB-JETS
!C   NSUB(J) =OUTPUT : NUMBER OF SUB-JETS AT YCUT(J)
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NY,NSUB(NY),NMAX,HIST,I,J,NUM
      PARAMETER (NMAX=512)
      DOUBLE PRECISION YCUT(NY),YMAC,ETOT,RSQ,P,KT,KTP,KTS,ETSQ,ECUT, &
          KTLAST,ROUND
      PARAMETER (ROUND=0.99999D0)
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
```

```
         IF (ETOT.EQ.0) CALL KTWARN('KTYSUB',100,*999)
         IF (ECUT.EQ.0) THEN
            ETSQ=1/ETOT**2
         ELSE
            ETSQ=1/ECUT**2
         ENDIF
         DO I=1,NY
            NSUB(I)=0
         END DO
         DO I=NUM,1,-1
            DO J=1,NY
               IF (NSUB(J).EQ.0.AND.KT(I)*ETSQ.GE.ROUND*YCUT(J)) NSUB(J)=I
               IF (NSUB(J).NE.0.AND.KTLAST(I)*ETSQ.LT.ROUND*YMAC) &
                     NSUB(J)=NSUB(J)-1
            END DO
         END DO
         RETURN
999      RETURN 1
      END SUBROUTINE KTYSUB
!C----------------------------------------------------------------------
      SUBROUTINE KTBEAM(ECUT,Y,*)
         IMPLICIT NONE
!C---GIVE SAME INFORMATION AS LAST CALL TO KTCLUS EXCEPT THAT ONLY
!C   TRANSITIONS WHERE A JET WAS MERGED WITH THE BEAM JET ARE RECORDED
!C
!C   ECUT    = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   Y(J)    =OUTPUT : Y VALUE WHERE Jth HARDEST JET WAS MERGED WITH BEAM
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
         INTEGER NMAX,HIST,NUM,I,J
         PARAMETER (NMAX=512)
         DOUBLE PRECISION ETOT,RSQ,P,KT,KTP,KTS,ECUT,ETSQ,Y(*),KTLAST
         COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
               KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
         IF (ETOT.EQ.0) CALL KTWARN('KTBEAM',100,*999)
         IF (ECUT.EQ.0) THEN
            ETSQ=1/ETOT**2
         ELSE
            ETSQ=1/ECUT**2
         ENDIF
         J=1
         DO I=1,NUM
            IF (HIST(I).LE.NMAX) THEN
               Y(J)=ETSQ*KT(I)
               J=J+1
            ENDIF
         END DO
         DO I=J,NUM
            Y(I)=0
         END DO
         RETURN
```

2279

```
999   RETURN 1
      END SUBROUTINE KTBEAM
!C----------------------------------------------------------------------
      SUBROUTINE KTJOIN(ECUT,YMAC,Y,*)
        IMPLICIT NONE
!C---GIVE SAME INFORMATION AS LAST CALL TO KTCLUS EXCEPT THAT ONLY
!C   TRANSITIONS WHERE TWO SUB-JETS WERE JOINED ARE RECORDED
!C   REMEMBER THAT A SUB-JET IS DEFINED AS A JET AT Y=YCUT WHICH HAS NOT
!C   YET BEEN MERGED WITH THE BEAM AT Y=YMAC.
!C
!C   ECUT   = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   YMAC   = INPUT : VALUE OF Y USED TO DEFINE MACRO-JETS
!C   Y(J)   =OUTPUT : Y VALUE WHERE EVENT CHANGED FROM HAVING
!C                         N+J SUB-JETS TO HAVING N+J-1, WHERE N IS
!C                         THE NUMBER OF MACRO-JETS AT SCALE YMAC
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,HIST,NUM,I,J
      PARAMETER (NMAX=512)
      DOUBLE PRECISION ETOT,RSQ,P,KT,KTP,KTS,ECUT,ETSQ,Y(*),YMAC,KTLAST, &
          ROUND
      PARAMETER (ROUND=0.99999D0)
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
      IF (ETOT.EQ.0) CALL KTWARN('KTJOIN',100,*999)
      IF (ECUT.EQ.0) THEN
         ETSQ=1/ETOT**2
      ELSE
         ETSQ=1/ECUT**2
      ENDIF
      J=1
      DO I=1,NUM
         IF (HIST(I).GT.NMAX.AND.ETSQ*KTLAST(I).GE.ROUND*YMAC) THEN
            Y(J)=ETSQ*KT(I)
            J=J+1
         ENDIF
      END DO
      DO I=J,NUM
         Y(I)=0
      END DO
      RETURN
999   RETURN 1
      END SUBROUTINE KTJOIN
!C----------------------------------------------------------------------
      SUBROUTINE KTRECO(RECO,PP,NN,ECUT,YCUT,YMAC,PJET,JET,NJET,NSUB,*)
        IMPLICIT NONE
!C---RECONSTRUCT KINEMATICS OF JET SYSTEM, WHICH HAS ALREADY BEEN
!C   ANALYSED BY KTCLUS. NOTE THAT NO CONSISTENCY CHECK IS MADE: USER
!C   IS TRUSTED TO USE THE SAME PP VALUES AS FOR KTCLUS
!C
!C   RECO   = INPUT : RECOMBINATION SCHEME (NEED NOT BE SAME AS KTCLUS)
```

```
!C    PP(I,J) = INPUT : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C    NN      = INPUT : NUMBER OF PARTICLES
!C    ECUT    = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C    YCUT    = INPUT : Y VALUE AT WHICH TO RECONSTRUCT JET MOMENTA
!C    YMAC    = INPUT : Y VALUE USED TO DEFINE MACRO-JETS, TO DETERMINE
!C                      WHICH JETS ARE SUB-JETS
!C    PJET(I,J)=OUTPUT : 4-MOMENTUM OF Jth JET AT SCALE YCUT
!C    JET(J)  =OUTPUT : THE MACRO-JET WHICH CONTAINS THE Jth JET,
!C                      SET TO ZERO IF JET IS NOT A SUB-JET
!C    NJET    =OUTPUT : THE NUMBER OF JETS
!C    NSUB    =OUTPUT : THE NUMBER OF SUB-JETS (EQUAL TO THE NUMBER OF
!C                      NON-ZERO ENTRIES IN JET())
!C    LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C    COULD NOT BE PROCESSED
!C
!C    NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C    AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,RECO,NUM,N,NN,NJET,NSUB,JET(*),HIST,IMIN,JMIN,I,J
      PARAMETER (NMAX=512)
      DOUBLE PRECISION PP(4,*),PJET(4,*)
      DOUBLE PRECISION ECUT,P,KT,KTP,KTS,ETOT,RSQ,ETSQ,YCUT,YMAC,KTLAST, &
         ROUND
      PARAMETER (ROUND=0.99999D0)
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
         KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
!C---CHECK INPUT
      if (signal_is_pending ()) return
      IF (RECO.LT.1.OR.RECO.GT.3) THEN
         PRINT *,'RECO=',RECO
         CALL KTWARN('KTRECO',100,*999)
      ENDIF
!C---COPY PP TO P
      N=NN
      IF (NUM.NE.NN) CALL KTWARN('KTRECO',101,*999)
      CALL KTCOPY(PP,N,P,(RECO.NE.1))
      IF (ECUT.EQ.0) THEN
         ETSQ=1/ETOT**2
      ELSE
         ETSQ=1/ECUT**2
      ENDIF
!C---KEEP MERGING UNTIL YCUT
 100  IF (ETSQ*KT(N).LT.ROUND*YCUT) THEN
         IF (HIST(N).LE.NMAX) THEN
            CALL KTMOVE(P,KTP,KTS,NMAX,N,HIST(N),0)
         ELSE
            IMIN=HIST(N)/NMAX
            JMIN=HIST(N)-IMIN*NMAX
            CALL KTMERG(P,KTP,KTS,NMAX,IMIN,JMIN,N,0,0,0,RECO)
            CALL KTMOVE(P,KTP,KTS,NMAX,N,JMIN,0)
         ENDIF
         N=N-1
         IF (N.GT.0) GOTO 100
      ENDIF
```

```
!C---IF YCUT IS TOO LARGE THERE ARE NO JETS
      NJET=N
      NSUB=N
      IF (N.EQ.0) RETURN
!C---SET UP OUTPUT MOMENTA
      DO I=1,NJET
         IF (RECO.EQ.1) THEN
            DO J=1,4
               PJET(J,I)=P(J,I)
            END DO
         ELSE
            PJET(1,I)=P(6,I)*COS(P(8,I))
            PJET(2,I)=P(6,I)*SIN(P(8,I))
            PJET(3,I)=P(6,I)*SINH(P(7,I))
            PJET(4,I)=P(6,I)*COSH(P(7,I))
         ENDIF
         JET(I)=I
      END DO
!C---KEEP MERGING UNTIL YMAC TO FIND THE FATE OF EACH JET
300   IF (ETSQ*KT(N).LT.ROUND*YMAC) THEN
         IF (HIST(N).LE.NMAX) THEN
            IMIN=0
            JMIN=HIST(N)
            NSUB=NSUB-1
         ELSE
            IMIN=HIST(N)/NMAX
            JMIN=HIST(N)-IMIN*NMAX
            IF (ETSQ*KTLAST(N).LT.ROUND*YMAC) NSUB=NSUB-1
         ENDIF
         DO I=1,NJET
            IF (JET(I).EQ.JMIN) JET(I)=IMIN
            IF (JET(I).EQ.N) JET(I)=JMIN
         END DO
         N=N-1
         IF (N.GT.0) GOTO 300
      ENDIF
      RETURN
999   RETURN 1
   END SUBROUTINE KTRECO
!C-------------------------------------------------------------------
   SUBROUTINE KTINCL(RECO,PP,NN,PJET,JET,NJET,*)
      IMPLICIT NONE
!C---RECONSTRUCT KINEMATICS OF JET SYSTEM, WHICH HAS ALREADY BEEN
!C   ANALYSED BY KTCLUS ACCORDING TO THE INCLUSIVE JET DEFINITION. NOTE
!C   THAT NO CONSISTENCY CHECK IS MADE: USER IS TRUSTED TO USE THE SAME
!C   PP VALUES AS FOR KTCLUS
!C
!C   RECO    = INPUT : RECOMBINATION SCHEME (NEED NOT BE SAME AS KTCLUS)
!C   PP(I,J) = INPUT : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C   NN      = INPUT : NUMBER OF PARTICLES
!C   PJET(I,J)=OUTPUT : 4-MOMENTUM OF Jth JET AT SCALE YCUT
!C   JET(J)  =OUTPUT : THE JET WHICH CONTAINS THE Jth PARTICLE
!C   NJET    =OUTPUT : THE NUMBER OF JETS
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
```

```fortran
!C    COULD NOT BE PROCESSED
!C
!C    NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C    AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,RECO,NUM,N,NN,NJET,JET(*),HIST,IMIN,JMIN,I,J
      PARAMETER (NMAX=512)
      DOUBLE PRECISION PP(4,*),PJET(4,*)
      DOUBLE PRECISION P,KT,KTP,KTS,ETOT,RSQ,KTLAST
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
!C---CHECK INPUT
      IF (RECO.LT.1.OR.RECO.GT.3) CALL KTWARN('KTINCL',100,*999)
!C---COPY PP TO P
      N=NN
      IF (NUM.NE.NN) CALL KTWARN('KTINCL',101,*999)
      CALL KTCOPY(PP,N,P,(RECO.NE.1))
!C---INITIALLY EVERY PARTICLE IS IN ITS OWN JET
      DO I=1,NN
         JET(I)=I
      END DO
!C---KEEP MERGING TO THE BITTER END
      NJET=0
200   IF (N.GT.0) THEN
         IF (HIST(N).LE.NMAX) THEN
            IMIN=0
            JMIN=HIST(N)
            NJET=NJET+1
            IF (RECO.EQ.1) THEN
               DO J=1,4
                  PJET(J,NJET)=P(J,JMIN)
               END DO
            ELSE
               PJET(1,NJET)=P(6,JMIN)*COS(P(8,JMIN))
               PJET(2,NJET)=P(6,JMIN)*SIN(P(8,JMIN))
               PJET(3,NJET)=P(6,JMIN)*SINH(P(7,JMIN))
               PJET(4,NJET)=P(6,JMIN)*COSH(P(7,JMIN))
            ENDIF
            CALL KTMOVE(P,KTP,KTS,NMAX,N,JMIN,0)
         ELSE
            IMIN=HIST(N)/NMAX
            JMIN=HIST(N)-IMIN*NMAX
            CALL KTMERG(P,KTP,KTS,NMAX,IMIN,JMIN,N,0,0,0,RECO)
            CALL KTMOVE(P,KTP,KTS,NMAX,N,JMIN,0)
         ENDIF
         DO I=1,NN
            IF (JET(I).EQ.JMIN) JET(I)=IMIN
            IF (JET(I).EQ.N) JET(I)=JMIN
            IF (JET(I).EQ.0) JET(I)=-NJET
         END DO
         N=N-1
         GOTO 200
      ENDIF
!C---FINALLY EVERY PARTICLE MUST BE IN AN INCLUSIVE JET
```

```
      DO I=1,NN
!C---IF THERE ARE ANY UNASSIGNED PARTICLES SOMETHING MUST HAVE GONE WRONG
        IF (JET(I).GE.0) CALL KTWARN('KTINCL',102,*999)
        JET(I)=-JET(I)
      END DO
      RETURN
 999  RETURN 1
    END SUBROUTINE KTINCL
!C-------------------------------------------------------------------
    SUBROUTINE KTISUB(N,NY,YCUT,NSUB,*)
      IMPLICIT NONE
!C---COUNT THE NUMBER OF SUB-JETS IN THE Nth INCLUSIVE JET OF AN EVENT
!C   THAT HAS ALREADY BEEN ANALYSED BY KTCLUS.
!C
!C   N       = INPUT : WHICH INCLUSIVE JET TO USE
!C   NY      = INPUT : NUMBER OF YCUT VALUES
!C   YCUT(J) = INPUT : Y VALUES AT WHICH NUMBERS OF SUB-JETS ARE COUNTED
!C   NSUB(J) =OUTPUT : NUMBER OF SUB-JETS AT YCUT(J)
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER N,NY,NSUB(NY),NMAX,HIST,I,J,NUM,NM
      PARAMETER (NMAX=512)
      DOUBLE PRECISION YCUT(NY),ETOT,RSQ,P,KT,KTP,KTS,KTLAST,ROUND,EPS
      PARAMETER (ROUND=0.99999D0)
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
      DATA EPS/1D-6/
      DO I=1,NY
         NSUB(I)=0
      END DO
!C---FIND WHICH MERGING CORRESPONDS TO THE NTH INCLUSIVE JET
      NM=0
      J=0
      DO I=NUM,1,-1
        IF (HIST(I).LE.NMAX) J=J+1
        IF (J.EQ.N) THEN
          NM=I
          GOTO 120
        ENDIF
      END DO
120   CONTINUE
!C---GIVE UP IF THERE ARE LESS THAN N INCLUSIVE JETS
      IF (NM.EQ.0) CALL KTWARN('KTISUB',100,*999)
      DO I=NUM,1,-1
         DO J=1,NY
            IF (NSUB(J).EQ.0.AND.RSQ*KT(I).GE.ROUND*YCUT(J)*KT(NM)) &
                NSUB(J)=I
            IF (NSUB(J).NE.0.AND.ABS(KTLAST(I)-KTLAST(NM)).GT.EPS) &
                NSUB(J)=NSUB(J)-1
         END DO
      END DO
```

```
          RETURN
999    RETURN 1
      END SUBROUTINE KTISUB
!C----------------------------------------------------------------------
      SUBROUTINE KTIJOI(N,Y,*)
        IMPLICIT NONE
!C---GIVE SAME INFORMATION AS LAST CALL TO KTCLUS EXCEPT THAT ONLY
!C   MERGES OF TWO SUB-JETS INSIDE THE Nth INCLUSIVE JET ARE RECORDED
!C
!C   N       = INPUT : WHICH INCLUSIVE JET TO USE
!C   Y(J)    =OUTPUT : Y VALUE WHERE JET CHANGED FROM HAVING
!C                         J+1 SUB-JETS TO HAVING J
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
        INTEGER NMAX,HIST,NUM,I,J,N,NM
        PARAMETER (NMAX=512)
        DOUBLE PRECISION ETOT,RSQ,P,KT,KTP,KTS,Y(*),KTLAST,EPS
        COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
            KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
        DATA EPS/1D-6/
!C---FIND WHICH MERGING CORRESPONDS TO THE NTH INCLUSIVE JET
      NM=0
      J=0
      DO I=NUM,1,-1
        IF (HIST(I).LE.NMAX) J=J+1
        IF (J.EQ.N) THEN
          NM=I
          GOTO 105
        ENDIF
      END DO
105   CONTINUE
!C---GIVE UP IF THERE ARE LESS THAN N INCLUSIVE JETS
      IF (NM.EQ.0) CALL KTWARN('KTIJOI',100,*999)
      J=1
      DO I=1,NUM
        IF (HIST(I).GT.NMAX.AND.ABS(KTLAST(I)-KTLAST(NM)).LT.EPS) THEN
          Y(J)=RSQ*KT(I)/KT(NM)
          J=J+1
        ENDIF
      END DO
      DO I=J,NUM
        Y(I)=0
      END DO
       RETURN
999    RETURN 1
      END SUBROUTINE KTIJOI
!C----------------------------------------------------------------------
      SUBROUTINE KTIREC(RECO,PP,NN,N,YCUT,PSUB,NSUB,*)
        IMPLICIT NONE
!C---RECONSTRUCT KINEMATICS OF SUB-JET SYSTEM IN THE Nth INCLUSIVE JET
!C   OF AN EVENT THAT HAS ALREADY BEEN ANALYSED BY KTCLUS
```

```
!C
!C   RECO    = INPUT : RECOMBINATION SCHEME (NEED NOT BE SAME AS KTCLUS)
!C   PP(I,J) = INPUT : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C   NN      = INPUT : NUMBER OF PARTICLES
!C   N       = INPUT : WHICH INCLUSIVE JET TO USE
!C   YCUT    = INPUT : Y VALUE AT WHICH TO RECONSTRUCT JET MOMENTA
!C   PSUB(I,J)=OUTPUT : 4-MOMENTUM OF Jth SUB-JET AT SCALE YCUT
!C   NSUB    =OUTPUT : THE NUMBER OF SUB-JETS
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C   AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,RECO,NUM,NN,NJET,NSUB,JET,HIST,I,J,N,NM
      PARAMETER (NMAX=512)
      DOUBLE PRECISION PP(4,*),PSUB(4,*)
      DOUBLE PRECISION ECUT,P,KT,KTP,KTS,ETOT,RSQ,YCUT,YMAC,KTLAST
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
      DIMENSION JET(NMAX)
!C---FIND WHICH MERGING CORRESPONDS TO THE NTH INCLUSIVE JET
      NM=0
      J=0
      DO I=NUM,1,-1
         IF (HIST(I).LE.NMAX) J=J+1
         IF (J.EQ.N) THEN
            NM=I
            GOTO 110
         ENDIF
      END DO
110   CONTINUE
!C---GIVE UP IF THERE ARE LESS THAN N INCLUSIVE JETS
      IF (NM.EQ.0) CALL KTWARN('KTIREC',102,*999)
!C---RECONSTRUCT THE JETS AT THE APPROPRIATE SCALE
      ECUT=SQRT(KT(NM)/RSQ)
      YMAC=RSQ
      CALL KTRECO(RECO,PP,NN,ECUT,YCUT,YMAC,PSUB,JET,NJET,NSUB,*999)
!C---GET RID OF THE ONES THAT DO NOT END UP IN THE JET WE WANT
      NSUB=0
      DO I=1,NJET
         IF (JET(I).EQ.HIST(NM)) THEN
            NSUB=NSUB+1
            DO J=1,4
               PSUB(J,NSUB)=PSUB(J,I)
            END DO
         ENDIF
      END DO
      RETURN
999   RETURN 1
      END SUBROUTINE KTIREC
!C---------------------------------------------------------------------
      SUBROUTINE KTWICH(ECUT,YCUT,JET,NJET,*)
        IMPLICIT NONE
```

```
!C---GIVE A LIST OF WHICH JET EACH ORIGINAL PARTICLE ENDED UP IN AT SCALE
!C    YCUT, TOGETHER WITH THE NUMBER OF JETS AT THAT SCALE.
!C
!C    ECUT     = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C    YCUT     = INPUT : Y VALUE AT WHICH TO DEFINE JETS
!C    JET(J)   =OUTPUT : THE JET WHICH CONTAINS THE Jth PARTICLE,
!C                              SET TO ZERO IF IT WAS PUT INTO THE BEAM JETS
!C    NJET     =OUTPUT : THE NUMBER OF JETS AT SCALE YCUT (SO JET()
!C                              ENTRIES WILL BE IN THE RANGE 0 -> NJET)
!C    LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C    COULD NOT BE PROCESSED
!C
!C    NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER JET(*),NJET,NTEMP
      DOUBLE PRECISION ECUT,YCUT
      CALL KTWCHS(ECUT,YCUT,YCUT,JET,NJET,NTEMP,*999)
      RETURN
 999  RETURN 1
    END SUBROUTINE KTWICH
!C--------------------------------------------------------------------
    SUBROUTINE KTWCHS(ECUT,YCUT,YMAC,JET,NJET,NSUB,*)
      IMPLICIT NONE
!C---GIVE A LIST OF WHICH SUB-JET EACH ORIGINAL PARTICLE ENDED UP IN AT
!C    SCALE YCUT, WITH MACRO-JET SCALE YMAC, TOGETHER WITH THE NUMBER OF
!C    JETS AT SCALE YCUT AND THE NUMBER OF THEM WHICH ARE SUB-JETS.
!C
!C    ECUT     = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C    YCUT     = INPUT : Y VALUE AT WHICH TO DEFINE JETS
!C    YMAC     = INPUT : Y VALUE AT WHICH TO DEFINE MACRO-JETS
!C    JET(J)   =OUTPUT : THE JET WHICH CONTAINS THE Jth PARTICLE,
!C                              SET TO ZERO IF IT WAS PUT INTO THE BEAM JETS
!C    NJET     =OUTPUT : THE NUMBER OF JETS AT SCALE YCUT (SO JET()
!C                              ENTRIES WILL BE IN THE RANGE 0 -> NJET)
!C    NSUB     =OUTPUT : THE NUMBER OF SUB-JETS AT SCALE YCUT, WITH
!C                              MACRO-JETS DEFINED AT SCALE YMAC (SO ONLY NSUB
!C                              OF THE JETS 1 -> NJET WILL APPEAR IN JET())
!C    LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C    COULD NOT BE PROCESSED
!C
!C    NOTE THAT ALL FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,JET(*),NJET,NSUB,HIST,NUM,I,J,JSUB
      PARAMETER (NMAX=512)
      DOUBLE PRECISION P1(4,NMAX),P2(4,NMAX)
      DOUBLE PRECISION ECUT,YCUT,YMAC,ZERO,ETOT,RSQ,P,KTP,KTS,KT,KTLAST
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
          KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
      DIMENSION JSUB(NMAX)
!C---THE MOMENTA HAVE TO BEEN GIVEN LEGAL VALUES,
!C    EVEN THOUGH THEY WILL NEVER BE USED
      DATA ((P1(J,I),I=1,NMAX),J=1,4),ZERO &
          /NMAX*1,NMAX*0,NMAX*0,NMAX*1,0/
!C---FIRST GET A LIST OF WHICH PARTICLE IS IN WHICH JET AT YCUT
```

2287

```
      CALL KTRECO(1,P1,NUM,ECUT,ZERO,YCUT,P2,JET,NJET,NSUB,*999)
!C---THEN FIND OUT WHICH JETS ARE SUBJETS
      CALL KTRECO(1,P1,NUM,ECUT,YCUT,YMAC,P2,JSUB,NJET,NSUB,*999)
!C---AND MODIFY JET() ACCORDINGLY
      DO I=1,NUM
         IF (JET(I).NE.0) THEN
            IF (JSUB(JET(I)).EQ.0) JET(I)=0
         ENDIF
      END DO
      RETURN
999   RETURN 1
   END SUBROUTINE KTWCHS
!C----------------------------------------------------------------------
   SUBROUTINE KTFRAM(IOPT,CMF,SIGN,Z,XZ,N,P,Q,*)
      IMPLICIT NONE
!C---BOOST PARTICLES IN P TO/FROM FRAME GIVEN BY CMF, Z, XZ.
!C---IN THIS FRAME CMZ IS STATIONARY,
!C                 Z IS ALONG THE (SIGN)Z-AXIS (SIGN=+ OR -)
!C                 XZ IS IN THE X-Z PLANE (WITH POSITIVE X COMPONENT)
!C---IF Z HAS LENGTH ZERO, OR SIGN=0, NO ROTATION IS PERFORMED
!C---IF XZ HAS ZERO COMPONENT PERPENDICULAR TO Z IN THAT FRAME,
!C   NO AZIMUTHAL ROTATION IS PERFORMED
!C
!C   IOPT    = INPUT : 0=TO FRAME, 1=FROM FRAME
!C   CMF(I)  = INPUT : 4-MOMENTUM WHICH IS STATIONARY IN THE FRAME
!C   SIGN    = INPUT : DIRECTION OF Z IN THE FRAME, NOTE THAT
!C                       ONLY ITS SIGN IS USED, NOT ITS MAGNITUDE
!C   Z(I)    = INPUT : 4-MOMENTUM WHICH LIES ON THE (SIGN)Z-AXIS
!C   XZ(I)   = INPUT : 4-MOMENTUM WHICH LIES IN THE X-Z PLANE
!C   N       = INPUT : NUMBER OF PARTICLES IN P
!C   P(I,J)  = INPUT : 4-MOMENTUM OF JTH PARTICLE BEFORE
!C   Q(I,J)  = OUTPUT : 4-MOMENTUM OF JTH PARTICLE AFTER
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT ALL MOMENTA ARE DOUBLE PRECISION
!C
!C   NOTE THAT IT IS SAFE TO CALL WITH P=Q
!C
      INTEGER IOPT,I,N
      DOUBLE PRECISION CMF(4),SIGN,Z(4),XZ(4),P(4,N),Q(4,N), &
         R(4,4),NEW(4),OLD(4)
      IF (IOPT.LT.0.OR.IOPT.GT.1) CALL KTWARN('KTFRAM',200,*999)
!C---FIND BOOST TO GET THERE FROM LAB
      CALL KTUNIT(R)
      CALL KTLBST(0,R,CMF,*999)
!C---FIND ROTATION TO PUT BOOSTED Z ON THE (SIGN)Z AXIS
      IF (SIGN.NE.0) THEN
         CALL KTVMUL(R,Z,OLD)
         IF (OLD(1).NE.0.OR.OLD(2).NE.0.OR.OLD(3).NE.0) THEN
            NEW(1)=0
            NEW(2)=0
            NEW(3)=SIGN
            NEW(4)=ABS(SIGN)
```

```
          CALL KTRROT(R,OLD,NEW,*999)
!C---FIND ROTATION TO PUT BOOSTED AND ROTATED XZ INTO X-Z PLANE
          CALL KTVMUL(R,XZ,OLD)
          IF (OLD(1).NE.0.OR.OLD(2).NE.0) THEN
             NEW(1)=1
             NEW(2)=0
             NEW(3)=0
             NEW(4)=1
             OLD(3)=0
!C---NOTE THAT A POTENTIALLY AWKWARD SPECIAL CASE IS AVERTED, BECAUSE IF
!C   OLD AND NEW ARE EXACTLY BACK-TO-BACK, THE ROTATION AXIS IS UNDEFINED
!C   BUT IN THAT CASE KTRROT WILL USE THE Z AXIS, AS REQUIRED
             CALL KTRROT(R,OLD,NEW,*999)
          ENDIF
        ENDIF
      ENDIF
!C---INVERT THE TRANSFORMATION IF NECESSARY
      IF (IOPT.EQ.1) CALL KTINVT(R,R)
!C---APPLY THE RESULT TO ALL THE VECTORS
      DO I=1,N
         CALL KTVMUL(R,P(1,I),Q(1,I))
      END DO
      RETURN
999   RETURN 1
    END SUBROUTINE KTFRAM
!C----------------------------------------------------------------------
    SUBROUTINE KTBREI(IOPT,PLEP,PHAD,POUT,N,P,Q,*)
      IMPLICIT NONE
!C---BOOST PARTICLES IN P TO/FROM BREIT FRAME
!C
!C   IOPT    = INPUT  : 0/2=TO BREIT FRAME, 1/3=FROM BREIT FRAME
!C                      0/1=NO AZIMUTHAL ROTATION AFTERWARDS
!C                      2/3=LEPTON PLANE ROTATED INTO THE X-Z PLANE
!C   PLEP    = INPUT  : MOMENTUM OF INCOMING LEPTON IN +Z DIRECTION
!C   PHAD    = INPUT  : MOMENTUM OF INCOMING HADRON IN +Z DIRECTION
!C   POUT(I) = INPUT  : 4-MOMENTUM OF OUTGOING LEPTON
!C   N       = INPUT  : NUMBER OF PARTICLES IN P
!C   P(I,J) = INPUT  : 4-MOMENTUM OF JTH PARTICLE BEFORE
!C   Q(I,J) = OUTPUT : 4-MOMENTUM OF JTH PARTICLE AFTER
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED (MOST LIKELY DUE TO PARTICLES HAVING SMALLER
!C   ENERGY THAN MOMENTUM)
!C
!C   NOTE THAT ALL MOMENTA ARE DOUBLE PRECISION
!C
!C   NOTE THAT IT IS SAFE TO CALL WITH P=Q
!C
      INTEGER IOPT,N
      DOUBLE PRECISION PLEP,PHAD,POUT(4),P(4,N),Q(4,N), &
           CMF(4),Z(4),XZ(4),DOT,QDQ
!C---CHECK INPUT
      IF (IOPT.LT.0.OR.IOPT.GT.3) CALL KTWARN('KTBREI',200,*999)
!C---FIND 4-MOMENTUM OF BREIT FRAME (TIMES AN ARBITRARY FACTOR)
      DOT=ABS(PHAD)*(ABS(PLEP)-POUT(4))-PHAD*(PLEP-POUT(3))
```

```fortran
      QDQ=(ABS(PLEP)-POUT(4))**2-(PLEP-POUT(3))**2-POUT(2)**2-POUT(1)**2
      CMF(1)=DOT*(           -POUT(1))
      CMF(2)=DOT*(           -POUT(2))
      CMF(3)=DOT*(    PLEP -POUT(3))-QDQ*     PHAD
      CMF(4)=DOT*(ABS(PLEP)-POUT(4))-QDQ*ABS(PHAD)
!C---FIND ROTATION TO PUT INCOMING HADRON BACK ON Z-AXIS
      Z(1)=0
      Z(2)=0
      Z(3)=PHAD
      Z(4)=ABS(PHAD)
      XZ(1)=0
      XZ(2)=0
      XZ(3)=0
      XZ(4)=0
!C---DO THE BOOST
      IF (IOPT.LE.1) THEN
         CALL KTFRAM(IOPT,CMF,PHAD,Z,XZ,N,P,Q,*999)
      ELSE
         CALL KTFRAM(IOPT-2,CMF,PHAD,Z,POUT,N,P,Q,*999)
      ENDIF
      RETURN
999   RETURN 1
   END SUBROUTINE KTBREI
!C---------------------------------------------------------------------
   SUBROUTINE KTHADR(IOPT,PLEP,PHAD,POUT,N,P,Q,*)
      IMPLICIT NONE
!C---BOOST PARTICLES IN P TO/FROM HADRONIC CMF
!C
!C   ARGUMENTS ARE EXACTLY AS FOR KTBREI
!C
!C   NOTE THAT ALL MOMENTA ARE DOUBLE PRECISION
!C
!C   NOTE THAT IT IS SAFE TO CALL WITH P=Q
!C
      INTEGER IOPT,N
      DOUBLE PRECISION PLEP,PHAD,POUT(4),P(4,N),Q(4,N), &
           CMF(4),Z(4),XZ(4)
!C---CHECK INPUT
      IF (IOPT.LT.0.OR.IOPT.GT.3) CALL KTWARN('KTHADR',200,*999)
!C---FIND 4-MOMENTUM OF HADRONIC CMF
      CMF(1)=           -POUT(1)
      CMF(2)=           -POUT(2)
      CMF(3)=     PLEP -POUT(3)+     PHAD
      CMF(4)=ABS(PLEP)-POUT(4)+ABS(PHAD)
!C---FIND ROTATION TO PUT INCOMING HADRON BACK ON Z-AXIS
      Z(1)=0
      Z(2)=0
      Z(3)=PHAD
      Z(4)=ABS(PHAD)
      XZ(1)=0
      XZ(2)=0
      XZ(3)=0
      XZ(4)=0
!C---DO THE BOOST
```

2290

```
          IF (IOPT.LE.1) THEN
              CALL KTFRAM(IOPT,CMF,PHAD,Z,XZ,N,P,Q,*999)
          ELSE
              CALL KTFRAM(IOPT-2,CMF,PHAD,Z,POUT,N,P,Q,*999)
          ENDIF
          RETURN
999   RETURN 1
      END SUBROUTINE KTHADR
!C----------------------------------------------------------------
      FUNCTION KTPAIR(ANGL,P,Q,ANGLE)
        IMPLICIT NONE
!C---CALCULATE LOCAL KT OF PAIR, USING ANGULAR SCHEME:
!C   1=>ANGULAR, 2=>DeltaR, 3=>f(DeltaEta,DeltaPhi)
!C   WHERE f(eta,phi)=2(COSH(eta)-COS(phi)) IS THE QCD EMISSION METRIC
!C---IF ANGLE<0, IT IS SET TO THE ANGULAR PART OF THE LOCAL KT ON RETURN
!C   IF ANGLE>0, IT IS USED INSTEAD OF THE ANGULAR PART OF THE LOCAL KT
        INTEGER ANGL,I
! CHANGE       DOUBLE PRECISION P(9),Q(9),KTPAIR,R,KTMDPI,ANGLE,ETA,PHI,ESQ
        DOUBLE PRECISION P(9),Q(9),KTPAIR,R,ANGLE,ETA,PHI,ESQ
!C---COMPONENTS OF MOMENTA ARE PX,PY,PZ,E,1/P,PT,ETA,PHI,PT**2
        R=ANGLE
        IF (ANGL.EQ.1) THEN
            IF (R.LE.0) R=2*(1-(P(1)*Q(1)+P(2)*Q(2)+P(3)*Q(3))*(P(5)*Q(5)))
            ESQ=MIN(P(4),Q(4))**2
        ELSEIF (ANGL.EQ.2.OR.ANGL.EQ.3) THEN
            IF (R.LE.0) THEN
                ETA=P(7)-Q(7)
                PHI=KTMDPI(P(8)-Q(8))
                IF (ANGL.EQ.2) THEN
                    R=ETA**2+PHI**2
                ELSE
                    R=2*(COSH(ETA)-COS(PHI))
                ENDIF
            ENDIF
            ESQ=MIN(P(9),Q(9))
        ELSEIF (ANGL.EQ.4) THEN
            ESQ=(1d0/(P(5)*Q(5))-P(1)*Q(1)-P(2)*Q(2)- &
                P(3)*Q(3))*2D0/(P(5)*Q(5))/(0.0001D0+1d0/P(5)+1d0/Q(5))**2
            R=1d0
        ELSE
            CALL KTWARN('KTPAIR',200,*999)
            STOP
        ENDIF
        KTPAIR=ESQ*R
        IF (ANGLE.LT.0) ANGLE=R
999 END FUNCTION KTPAIR
!C----------------------------------------------------------------
      FUNCTION KTSING(ANGL,TYPE,P)
        IMPLICIT NONE
!C---CALCULATE KT OF PARTICLE, USING ANGULAR SCHEME:
!C   1=>ANGULAR, 2=>DeltaR, 3=>f(DeltaEta,DeltaPhi)
!C---TYPE=1 FOR E+E-, 2 FOR EP, 3 FOR PE, 4 FOR PP
!C   FOR EP, PROTON DIRECTION IS DEFINED AS -Z
!C   FOR PE, PROTON DIRECTION IS DEFINED AS +Z
```

2291

```
      INTEGER ANGL,TYPE
      DOUBLE PRECISION P(9),KTSING,COSTH,R,SMALL
      DATA SMALL/1D-4/
      IF (ANGL.EQ.1.OR.ANGL.EQ.4) THEN
          COSTH=P(3)*P(5)
          IF (TYPE.EQ.2) THEN
              COSTH=-COSTH
          ELSEIF (TYPE.EQ.4) THEN
              COSTH=ABS(COSTH)
          ELSEIF (TYPE.NE.1.AND.TYPE.NE.3) THEN
              CALL KTWARN('KTSING',200,*999)
              STOP
          ENDIF
          R=2*(1-COSTH)
!C---IF CLOSE TO BEAM, USE APPROX 2*(1-COS(THETA))=SIN**2(THETA)
          IF (R.LT.SMALL) R=(P(1)**2+P(2)**2)*P(5)**2
          KTSING=P(4)**2*R
      ELSEIF (ANGL.EQ.2.OR.ANGL.EQ.3) THEN
          KTSING=P(9)
      ELSE
          CALL KTWARN('KTSING',201,*999)
          STOP
      ENDIF
999 END FUNCTION KTSING
!C----------------------------------------------------------------
      SUBROUTINE KTPMIN(A,NMAX,N,IMIN,JMIN)
        IMPLICIT NONE
!C---FIND THE MINIMUM MEMBER OF A(NMAX,NMAX) WITH IMIN < JMIN <= N
      INTEGER NMAX,N,IMIN,JMIN,KMIN,I,J,K
!C---REMEMBER THAT A(X+(Y-1)*NMAX)=A(X,Y)
!C   THESE LOOPING VARIABLES ARE J=Y-2, I=X+(Y-1)*NMAX
      DOUBLE PRECISION A(*),AMIN
      K=1+NMAX
      KMIN=K
      AMIN=A(KMIN)
      DO J=0,N-2
          DO I=K,K+J
              IF (A(I).LT.AMIN) THEN
                  KMIN=I
                  AMIN=A(KMIN)
              ENDIF
          END DO
          K=K+NMAX
      END DO
      JMIN=KMIN/NMAX+1
      IMIN=KMIN-(JMIN-1)*NMAX
    END SUBROUTINE KTPMIN
!C----------------------------------------------------------------
    SUBROUTINE KTSMIN(A,NMAX,N,IMIN)
      IMPLICIT NONE
!C---FIND THE MINIMUM MEMBER OF A
      INTEGER N,NMAX,IMIN,I
      DOUBLE PRECISION A(NMAX)
      IMIN=1
```

```
      DO I=1,N
         IF (A(I).LT.A(IMIN)) IMIN=I
      END DO
   END SUBROUTINE KTSMIN
!C----------------------------------------------------------------------
   SUBROUTINE KTCOPY(A,N,B,ONSHLL)
      IMPLICIT NONE
!C---COPY FROM A TO B. 5TH=1/(3-MTM), 6TH=PT, 7TH=ETA, 8TH=PHI, 9TH=PT**2
!C   IF ONSHLL IS .TRUE. PARTICLE ENTRIES ARE PUT ON-SHELL BY SETTING E=P
      INTEGER I,N
      DOUBLE PRECISION A(4,N)
      LOGICAL ONSHLL
      DOUBLE PRECISION B(9,N),ETAMAX,SINMIN,EPS
      DATA ETAMAX,SINMIN,EPS/10,0,1D-6/
!C---SINMIN GETS CALCULATED ON FIRST CALL
      IF (SINMIN.EQ.0) SINMIN=1/COSH(ETAMAX)
      DO I=1,N
         B(1,I)=A(1,I)
         B(2,I)=A(2,I)
         B(3,I)=A(3,I)
         B(4,I)=A(4,I)
         B(5,I)=SQRT(A(1,I)**2+A(2,I)**2+A(3,I)**2)
         IF (ONSHLL) B(4,I)=B(5,I)
         IF (B(5,I).EQ.0) B(5,I)=1D-10
         B(5,I)=1/B(5,I)
         B(9,I)=A(1,I)**2+A(2,I)**2
         B(6,I)=SQRT(B(9,I))
         B(7,I)=B(6,I)*B(5,I)
         IF (B(7,I).GT.SINMIN) THEN
            B(7,I)=A(4,I)**2-A(3,I)**2
            IF (B(7,I).LE.EPS*B(4,I)**2.OR.ONSHLL) B(7,I)=B(9,I)
            B(7,I)=LOG((B(4,I)+ABS(B(3,I)))**2/B(7,I))/2
         ELSE
            B(7,I)=ETAMAX+2
         ENDIF
         B(7,I)=SIGN(B(7,I),B(3,I))
         IF (A(1,I).EQ.0 .AND. A(2,I).EQ.0) THEN
            B(8,I)=0
         ELSE
            B(8,I)=ATAN2(A(2,I),A(1,I))
         ENDIF
      END DO
   END SUBROUTINE KTCOPY
!C----------------------------------------------------------------------
   SUBROUTINE KTMERG(P,KTP,KTS,NMAX,I,J,N,TYPE,ANGL,MONO,RECO)
      IMPLICIT NONE
!C---MERGE THE Jth PARTICLE IN P INTO THE Ith PARTICLE
!C   J IS ASSUMED GREATER THAN I. P CONTAINS N PARTICLES BEFORE MERGING.
!C---ALSO RECALCULATING THE CORRESPONDING KTP AND KTS VALUES IF MONO.GT.0
!C   FROM THE RECOMBINED ANGULAR MEASURES IF MONO.GT.1
!C---NOTE THAT IF MONO.LE.0, TYPE AND ANGL ARE NOT USED
      INTEGER ANGL,RECO,TYPE,I,J,K,N,NMAX,MONO
      DOUBLE PRECISION P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX),PT,PTT, &
! CHANGE            KTMDPI,KTUP,PI,PJ,ANG,KTPAIR,KTSING,ETAMAX,EPS
```

2293

```
          KTUP,PI,PJ,ANG,ETAMAX,EPS
      KTUP(I,J)=KTP(MAX(I,J),MIN(I,J))
      DATA ETAMAX,EPS/10,1D-6/
      IF (J.LE.I) CALL KTWARN('KTMERG',200,*999)
!C---COMBINE ANGULAR MEASURES IF NECESSARY
      IF (MONO.GT.1) THEN
          DO K=1,N
              IF (K.NE.I.AND.K.NE.J) THEN
                  IF (RECO.EQ.1) THEN
                      PI=P(4,I)
                      PJ=P(4,J)
                  ELSEIF (RECO.EQ.2) THEN
                      PI=P(6,I)
                      PJ=P(6,J)
                  ELSEIF (RECO.EQ.3) THEN
                      PI=P(9,I)
                      PJ=P(9,J)
                  ELSE
                      CALL KTWARN('KTMERG',201,*999)
                      STOP
                  ENDIF
                  IF (PI.EQ.0.AND.PJ.EQ.0) THEN
                      PI=1
                      PJ=1
                  ENDIF
                  KTP(MAX(I,K),MIN(I,K))= &
                      (PI*KTUP(I,K)+PJ*KTUP(J,K))/(PI+PJ)
              ENDIF
          END DO
      ENDIF
      IF (RECO.EQ.1) THEN
!C---VECTOR ADDITION
          P(1,I)=P(1,I)+P(1,J)
          P(2,I)=P(2,I)+P(2,J)
          P(3,I)=P(3,I)+P(3,J)
!c        P(4,I)=P(4,I)+P(4,J) ! JA
          P(5,I)=SQRT(P(1,I)**2+P(2,I)**2+P(3,I)**2)
          P(4,I)=P(5,I) ! JA (Massless scheme)
          IF (P(5,I).EQ.0) THEN
              P(5,I)=1
          ELSE
              P(5,I)=1/P(5,I)
          ENDIF
      ELSEIF (RECO.EQ.2) THEN
!C---PT WEIGHTED ETA-PHI ADDITION
          PT=P(6,I)+P(6,J)
          IF (PT.EQ.0) THEN
              PTT=1
          ELSE
              PTT=1/PT
          ENDIF
          P(7,I)=(P(6,I)*P(7,I)+P(6,J)*P(7,J))*PTT
          P(8,I)=KTMDPI(P(8,I)+P(6,J)*PTT*KTMDPI(P(8,J)-P(8,I)))
          P(6,I)=PT
```

```
            P(9,I)=PT**2
        ELSEIF (RECO.EQ.3) THEN
!C---PT**2 WEIGHTED ETA-PHI ADDITION
            PT=P(9,I)+P(9,J)
            IF (PT.EQ.0) THEN
                PTT=1
            ELSE
                PTT=1/PT
            ENDIF
            P(7,I)=(P(9,I)*P(7,I)+P(9,J)*P(7,J))*PTT
            P(8,I)=KTMDPI(P(8,I)+P(9,J)*PTT*KTMDPI(P(8,J)-P(8,I)))
            P(6,I)=P(6,I)+P(6,J)
            P(9,I)=P(6,I)**2
        ELSE
            CALL KTWARN('KTMERG',202,*999)
            STOP
        ENDIF
!C---IF MONO.GT.0 CALCULATE NEW KT MEASURES. IF MONO.GT.1 USE ANGULAR ONES.
        IF (MONO.LE.0) RETURN
!C---CONVERTING BETWEEN 4-MTM AND PT,ETA,PHI IF NECESSARY
        IF (ANGL.NE.1.AND.RECO.EQ.1) THEN
            P(9,I)=P(1,I)**2+P(2,I)**2
            P(7,I)=P(4,I)**2-P(3,I)**2
            IF (P(7,I).LE.EPS*P(4,I)**2) P(7,I)=P(9,I)
            IF (P(7,I).GT.0) THEN
                P(7,I)=LOG((P(4,I)+ABS(P(3,I)))**2/P(7,I))/2
                IF (P(7,I).GT.ETAMAX) P(7,I)=ETAMAX+2
            ELSE
                P(7,I)=ETAMAX+2
            ENDIF
            P(7,I)=SIGN(P(7,I),P(3,I))
            IF (P(1,I).NE.0.AND.P(2,I).NE.0) THEN
                P(8,I)=ATAN2(P(2,I),P(1,I))
            ELSE
                P(8,I)=0
            ENDIF
        ELSEIF (ANGL.EQ.1.AND.RECO.NE.1) THEN
            P(1,I)=P(6,I)*COS(P(8,I))
            P(2,I)=P(6,I)*SIN(P(8,I))
            P(3,I)=P(6,I)*SINH(P(7,I))
            P(4,I)=P(6,I)*COSH(P(7,I))
            IF (P(4,I).NE.0) THEN
                P(5,I)=1/P(4,I)
            ELSE
                P(5,I)=1
            ENDIF
        ENDIF
        ANG=0
        DO K=1,N
            IF (K.NE.I.AND.K.NE.J) THEN
                IF (MONO.GT.1) ANG=KTUP(I,K)
                KTP(MIN(I,K),MAX(I,K))= &
                        KTPAIR(ANGL,P(1,I),P(1,K),ANG)
            ENDIF
```

```
          END DO
          KTS(I)=KTSING(ANGL,TYPE,P(1,I))
999 END SUBROUTINE KTMERG
!C----------------------------------------------------------------------
      SUBROUTINE KTMOVE(P,KTP,KTS,NMAX,N,J,IOPT)
        IMPLICIT NONE
!C---MOVE THE Nth PARTICLE IN P TO THE Jth POSITION
!C---ALSO MOVING KTP AND KTS IF IOPT.GT.0
        INTEGER I,J,N,NMAX,IOPT
        DOUBLE PRECISION P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX)
        DO I=1,9
           P(I,J)=P(I,N)
        END DO
        IF (IOPT.LE.0) RETURN
        DO I=1,J-1
           KTP(I,J)=KTP(I,N)
           KTP(J,I)=KTP(N,I)
        END DO
        DO I=J+1,N-1
           KTP(J,I)=KTP(I,N)
           KTP(I,J)=KTP(N,I)
        END DO
        KTS(J)=KTS(N)
      END SUBROUTINE KTMOVE
!C----------------------------------------------------------------------
      SUBROUTINE KTUNIT(R)
        IMPLICIT NONE
!C  SET R EQUAL TO THE 4 BY 4 IDENTITY MATRIX
        DOUBLE PRECISION R(4,4)
        INTEGER I,J
        DO I=1,4
           DO J=1,4
              R(I,J)=0
              IF (I.EQ.J) R(I,J)=1
           END DO
        END DO
      END SUBROUTINE KTUNIT
!C----------------------------------------------------------------------
      SUBROUTINE KTLBST(IOPT,R,A,*)
        IMPLICIT NONE
!C  PREMULTIPLY R BY THE 4 BY 4 MATRIX TO
!C  LORENTZ BOOST TO/FROM THE CM FRAME OF A
!C  IOPT=0 => TO
!C  IOPT=1 => FROM
!C
!C  LAST ARGUMENT IS LABEL TO JUMP TO IF A IS NOT TIME-LIKE
!C
        INTEGER IOPT,I,J
        DOUBLE PRECISION R(4,4),A(4),B(4),C(4,4),M
        DO I=1,4
           B(I)=A(I)
        END DO
        M=B(4)**2-B(1)**2-B(2)**2-B(3)**2
        IF (M.LE.0) CALL KTWARN('KTLBST',100,*999)
```

2296

```
        M=SQRT(M)
        B(4)=B(4)+M
        M=1/(M*B(4))
        IF (IOPT.EQ.0) THEN
          B(4)=-B(4)
        ELSEIF (IOPT.NE.1) THEN
          CALL KTWARN('KTLBST',200,*999)
          STOP
        ENDIF
        DO I=1,4
          DO J=1,4
            C(I,J)=B(I)*B(J)*M
            IF (I.EQ.J) C(I,J)=C(I,J)+1
          END DO
        END DO
        C(4,4)=C(4,4)-2
        CALL KTMMUL(C,R,R)
        RETURN
999     RETURN 1
      END SUBROUTINE KTLBST
!C----------------------------------------------------------------
      SUBROUTINE KTRROT(R,A,B,*)
        IMPLICIT NONE
!C   PREMULTIPLY R BY THE 4 BY 4 MATRIX TO
!C   ROTATE FROM VECTOR A TO VECTOR B BY THE SHORTEST ROUTE
!C   IF THEY ARE EXACTLY BACK-TO-BACK, THE ROTATION AXIS IS THE VECTOR
!C   WHICH IS PERPENDICULAR TO THEM AND THE X AXIS, UNLESS THEY ARE
!C   PERPENDICULAR TO THE Y AXIS, WHEN IT IS THE VECTOR WHICH IS
!C   PERPENDICULAR TO THEM AND THE Y AXIS.
!C   NOTE THAT THESE CONDITIONS GUARANTEE THAT IF BOTH ARE PERPENDICULAR
!C   TO THE Z AXIS, IT WILL BE USED AS THE ROTATION AXIS.
!C
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF EITHER HAS LENGTH ZERO
!C
        DOUBLE PRECISION R(4,4),M(4,4),A(4),B(4),C(4),D(4),AL,BL,CL,DL,EPS
!C---SQRT(2*EPS) IS THE ANGLE IN RADIANS OF THE SMALLEST ALLOWED ROTATION
!C   NOTE THAT IF YOU CONVERT THIS PROGRAM TO SINGLE PRECISION, YOU WILL
!C   NEED TO INCREASE EPS TO AROUND 0.5E-4
        PARAMETER (EPS=0.5D-6)
        AL=A(1)**2+A(2)**2+A(3)**2
        BL=B(1)**2+B(2)**2+B(3)**2
        IF (AL.LE.0.OR.BL.LE.0) CALL KTWARN('KTRROT',100,*999)
        AL=1/SQRT(AL)
        BL=1/SQRT(BL)
        CL=(A(1)*B(1)+A(2)*B(2)+A(3)*B(3))*AL*BL
!C---IF THEY ARE COLLINEAR, DON'T NEED TO DO ANYTHING
        IF (CL.GE.1-EPS) THEN
          RETURN
!C---IF THEY ARE BACK-TO-BACK, USE THE AXIS PERP TO THEM AND X AXIS
        ELSEIF (CL.LE.-1+EPS) THEN
          IF (ABS(B(2)).GT.EPS) THEN
            C(1)= 0
            C(2)=-B(3)
            C(3)= B(2)
```

```
!C---UNLESS THEY ARE PERPENDICULAR TO THE Y AXIS,
      ELSE
          C(1)= B(3)
          C(2)= 0
          C(3)=-B(1)
      ENDIF
!C---OTHERWISE FIND ROTATION AXIS
    ELSE
        C(1)=A(2)*B(3)-A(3)*B(2)
        C(2)=A(3)*B(1)-A(1)*B(3)
        C(3)=A(1)*B(2)-A(2)*B(1)
    ENDIF
    CL=C(1)**2+C(2)**2+C(3)**2
    IF (CL.LE.0) CALL KTWARN('KTRROT',101,*999)
    CL=1/SQRT(CL)
!C---FIND ROTATION TO INTERMEDIATE AXES FROM A
    D(1)=A(2)*C(3)-A(3)*C(2)
    D(2)=A(3)*C(1)-A(1)*C(3)
    D(3)=A(1)*C(2)-A(2)*C(1)
    DL=AL*CL
    M(1,1)=A(1)*AL
    M(1,2)=A(2)*AL
    M(1,3)=A(3)*AL
    M(1,4)=0
    M(2,1)=C(1)*CL
    M(2,2)=C(2)*CL
    M(2,3)=C(3)*CL
    M(2,4)=0
    M(3,1)=D(1)*DL
    M(3,2)=D(2)*DL
    M(3,3)=D(3)*DL
    M(3,4)=0
    M(4,1)=0
    M(4,2)=0
    M(4,3)=0
    M(4,4)=1
    CALL KTMMUL(M,R,R)
!C---AND ROTATION FROM INTERMEDIATE AXES TO B
    D(1)=B(2)*C(3)-B(3)*C(2)
    D(2)=B(3)*C(1)-B(1)*C(3)
    D(3)=B(1)*C(2)-B(2)*C(1)
    DL=BL*CL
    M(1,1)=B(1)*BL
    M(2,1)=B(2)*BL
    M(3,1)=B(3)*BL
    M(1,2)=C(1)*CL
    M(2,2)=C(2)*CL
    M(3,2)=C(3)*CL
    M(1,3)=D(1)*DL
    M(2,3)=D(2)*DL
    M(3,3)=D(3)*DL
    CALL KTMMUL(M,R,R)
    RETURN
999 RETURN 1
```

```
      END SUBROUTINE KTRROT
!C----------------------------------------------------------------------
      SUBROUTINE KTVMUL(M,A,B)
        IMPLICIT NONE
!C   4 BY 4 MATRIX TIMES 4 VECTOR: B=M*A.
!C   ALL ARE DOUBLE PRECISION
!C   IT IS SAFE TO CALL WITH B=A
!C   FIRST SUBSCRIPT=ROWS, SECOND=COLUMNS
        DOUBLE PRECISION M(4,4),A(4),B(4),C(4)
        INTEGER I,J
        DO I=1,4
           C(I)=0
           DO J=1,4
              C(I)=C(I)+M(I,J)*A(J)
           END DO
        END DO
        DO I=1,4
           B(I)=C(I)
        END DO
      END SUBROUTINE KTVMUL
!C----------------------------------------------------------------------
      SUBROUTINE KTMMUL(A,B,C)
        IMPLICIT NONE
!C   4 BY 4 MATRIX MULTIPLICATION: C=A*B.
!C   ALL ARE DOUBLE PRECISION
!C   IT IS SAFE TO CALL WITH C=A OR B.
!C   FIRST SUBSCRIPT=ROWS, SECOND=COLUMNS
        DOUBLE PRECISION A(4,4),B(4,4),C(4,4),D(4,4)
        INTEGER I,J,K
        DO I=1,4
           DO J=1,4
              D(I,J)=0
              DO K=1,4
                 D(I,J)=D(I,J)+A(I,K)*B(K,J)
              END DO
           END DO
        END DO
        DO I=1,4
           DO J=1,4
              C(I,J)=D(I,J)
           END DO
        END DO
      END SUBROUTINE KTMMUL
!C----------------------------------------------------------------------
      SUBROUTINE KTINVT(A,B)
        IMPLICIT NONE
!C---INVERT TRANSFORMATION MATRIX A
!C
!C   A = INPUT  : 4 BY 4 TRANSFORMATION MATRIX
!C   B = OUTPUT : INVERTED TRANSFORMATION MATRIX
!C
!C   IF A IS NOT A TRANSFORMATION MATRIX YOU WILL GET STRANGE RESULTS
!C
!C   NOTE THAT IT IS SAFE TO CALL WITH A=B
```

```
!C
      DOUBLE PRECISION A(4,4),B(4,4),C(4,4)
      INTEGER I,J
!C---TRANSPOSE
      DO I=1,4
         DO J=1,4
            C(I,J)=A(J,I)
         END DO
      END DO
!C---NEGATE ENERGY-MOMENTUM MIXING TERMS
      DO I=1,3
         C(4,I)=-C(4,I)
         C(I,4)=-C(I,4)
      END DO
!C---OUTPUT
      DO I=1,4
         DO J=1,4
            B(I,J)=C(I,J)
         END DO
      END DO
    END SUBROUTINE KTINVT
!C----------------------------------------------------------------------
    FUNCTION KTMDPI(PHI)
      IMPLICIT NONE
!C---RETURNS PHI, MOVED ONTO THE RANGE [-PI,PI)
      DOUBLE PRECISION KTMDPI,PHI,PI,TWOPI,THRPI,EPS
      PARAMETER (PI=3.14159265358979324D0,TWOPI=6.28318530717958648D0, &
           THRPI=9.42477796076937972D0)
      PARAMETER (EPS=1D-15)
      KTMDPI=PHI
      IF (KTMDPI.LE.PI) THEN
        IF (KTMDPI.GT.-PI) THEN
          GOTO 100
        ELSEIF (KTMDPI.GT.-THRPI) THEN
          KTMDPI=KTMDPI+TWOPI
        ELSE
          KTMDPI=-MOD(PI-KTMDPI,TWOPI)+PI
        ENDIF
      ELSEIF (KTMDPI.LE.THRPI) THEN
        KTMDPI=KTMDPI-TWOPI
      ELSE
        KTMDPI=MOD(PI+KTMDPI,TWOPI)-PI
      ENDIF
 100  IF (ABS(KTMDPI).LT.EPS) KTMDPI=0
    END FUNCTION KTMDPI
!C----------------------------------------------------------------------
    SUBROUTINE KTWARN(SUBRTN,ICODE,*)
!C    DEALS WITH ERRORS DURING EXECUTION
!C    SUBRTN = NAME OF CALLING SUBROUTINE
!C    ICODE  = ERROR CODE:   - 99 PRINT WARNING & CONTINUE
!C                          100-199 PRINT WARNING & JUMP
!C                          200-    PRINT WARNING & STOP DEAD
!C----------------------------------------------------------------------
      INTEGER ICODE
```

```
      CHARACTER*6 SUBRTN
      WRITE (6,10) SUBRTN,ICODE
10    FORMAT(/' KTWARN CALLED FROM SUBPROGRAM ',A6,': CODE =',I4/)
      IF (ICODE.LT.100) RETURN
      IF (ICODE.LT.200) RETURN 1
      STOP
    END SUBROUTINE KTWARN
!C----------------------------------------------------------------------
!C----------------------------------------------------------------------
!C----------------------------------------------------------------------
```

16.5.2 CKKW matching

⟨ckkw_matching.f90⟩≡
  ⟨File header⟩

```
  module ckkw_matching
```

  ⟨Use kinds⟩
```
    use kinds, only: double !NODEP!
    use constants !NODEP!
    use lorentz !NODEP!
    use file_utils !NODEP!
    use diagnostics !NODEP!
    use tao_random_numbers !NODEP!
    use shower_base
    use shower_partons
    use shower_core
    use ckkw_pseudo_weights
```

  ⟨Standard module head⟩

  ⟨CKKW matching: public⟩

  ⟨CKKW matching: types⟩

```
  contains
```

  ⟨CKKW matching: procedures⟩

```
  end module ckkw_matching
```

⟨CKKW matching: public⟩≡
```
    public :: ckkw_matching_settings_t
```
⟨CKKW matching: types⟩≡
```
    type :: ckkw_matching_settings_t
       real(default) :: alphaS = 0.118_default
       real(default) :: Qmin = one
       integer :: n_max_jets = 0
    end type ckkw_matching_settings_t
```

⟨CKKW matching: public⟩+≡
```
    public :: ckkw_matching_apply
```

2301

⟨*CKKW matching: procedures*⟩≡

```
subroutine ckkw_matching_apply (shower, settings, weights, veto)
  type(shower_t), intent(inout) :: shower
  type(ckkw_matching_settings_t), intent(in) :: settings
  type(ckkw_pseudo_shower_weights_t), intent(in) :: weights
  logical, intent(out) :: veto

  real(default), dimension(:), allocatable :: scales
  real(double) :: weight, sf
  real(default) :: rand
  integer :: i

  if (signal_is_pending ()) return
  weight = 1.0

  call shower%write ()

  !!! the pseudo parton shower is already simulated by shower_add_interaction
  !!! get the respective clustering scales
  allocate (scales(1:size(shower%partons)))
  do i = 1, size (shower%partons)
     if (.not. associated (shower%partons(i)%p)) cycle
     if (shower%partons(i)%p%type == 94) then
        scales(i) = 2.0 * min (parton_get_energy (shower%partons(i)%p%child1),  &
              parton_get_energy (shower%partons(i)%p%child2))**2 * &
              (1.0 - (space_part (shower%partons(i)%p%child1%momentum) * &
               space_part (shower%partons(i)%p%child2%momentum)) / &
              (space_part (shower%partons(i)%p%child1%momentum)**1 * &
               space_part (shower%partons(i)%p%child2%momentum)**1))
        scales(i) = sqrt (scales(i))
        shower%partons(i)%p%ckkwscale = scales(i)
        print *, scales(i)
     end if
  end do

  print *, " scales finished"
  !!! if (highest multiplicity) -> reweight with PDF(mu_F) / PDF(mu_cut)
  call shower%write ()

  !!! Reweight and possibly veto the whole event

  !!! calculate the relative alpha_S weight

  !! calculate the Sudakov weights for internal lines
  !! calculate the Sudakov weights for external lines
  do i = 1, size (shower%partons)
     if (signal_is_pending ()) return
     if (.not. associated (shower%partons(i)%p)) cycle
     if (shower%partons(i)%p%type == 94) then
        !!! get type
        !!! check that all particles involved are colored
        if ((parton_is_colored (shower%partons(i)%p) .or. &
              shower%partons(i)%p%ckkwtype > 0) .and. &
              (parton_is_colored (shower%partons(i)%p%child1) .or. &
```

2302

```fortran
           shower%partons(i)%p%child1%ckkwtype > 0) .and. &
           (parton_is_colored (shower%partons(i)%p%child1) .or. &
           shower%partons(i)%p%child1%ckkwtype > 0)) then
        print *, "reweight with alphaS(" , shower%partons(i)%p%ckkwscale, &
             ") for particle ", shower%partons(i)%p%nr
        if (shower%partons(i)%p%belongstoFSR) then
           print *, "FSR"
           weight = weight * D_alpha_s_fsr (shower%partons(i)%p%ckkwscale**2) &
                / settings%alphas
        else
           print *, "ISR"
           weight = weight * &
                D_alpha_s_isr (shower%partons(i)%p%ckkwscale**2) &
                / settings%alphas
        end if
     else
        print *, "no reweight with alphaS for ", shower%partons(i)%p%nr
     end if
     if (shower%partons(i)%p%child1%type == 94) then
        print *, "internal line from ", &
             shower%partons(i)%p%child1%ckkwscale, &
             " to ", shower%partons(i)%p%ckkwscale, &
             " for type ", shower%partons(i)%p%child1%ckkwtype
        if (shower%partons(i)%p%child1%ckkwtype == 0) then
           sf = 1.0
        else if (shower%partons(i)%p%child1%ckkwtype == 1) then
           sf = SudakovQ (shower%partons(i)%p%child1%ckkwscale, &
                shower%partons(i)%p%ckkwscale, .true.)
           print *, "SFQ = ", sf
        else if (shower%partons(i)%p%child1%ckkwtype == 2) then
           sf = SudakovG (shower%partons(i)%p%child1%ckkwscale, &
                shower%partons(i)%p%ckkwscale, .true.)
           print *, "SFG = ", sf
        else
           print *, "SUSY not yet implemented"
        end if
        weight = weight * min (one, sf)
     else
        print *, "external line from ", settings%Qmin, &
             shower%partons(i)%p%ckkwscale
        if (parton_is_quark (shower%partons(i)%p%child1)) then
           sf = SudakovQ (settings%Qmin, &
                shower%partons(i)%p%ckkwscale, .true.)
           print *, "SFQ = ", sf
        else if (parton_is_gluon (shower%partons(i)%p%child1)) then
           sf = SudakovG (settings%Qmin, &
                shower%partons(i)%p%ckkwscale, .true.)
           print *, "SFG = ", sf
        else
           print *, "not yet implemented (", &
                shower%partons(i)%p%child2%type, ")"
           sf = one
        end if
        weight = weight * min (one, sf)
```

```fortran
            end if
            if (shower%partons(i)%p%child2%type == 94) then
               print *, "internal line from ", shower%partons(i)%p%child2%ckkwscale, &
                    " to ", shower%partons(i)%p%ckkwscale, &
                    " for type ", shower%partons(i)%p%child2%ckkwtype
               if (shower%partons(i)%p%child2%ckkwtype == 0) then
                  sf = 1.0
               else if (shower%partons(i)%p%child2%ckkwtype == 1) then
                  sf = SudakovQ (shower%partons(i)%p%child2%ckkwscale, &
                       shower%partons(i)%p%ckkwscale, .true.)
                  print *, "SFQ = ", sf
               else if (shower%partons(i)%p%child2%ckkwtype == 2) then
                  sf = SudakovG (shower%partons(i)%p%child2%ckkwscale, &
                       shower%partons(i)%p%ckkwscale, .true.)
                  print *, "SFG = ", sf
               else
                  print *, "SUSY not yet implemented"
               end if
               weight = weight * min (one, sf)
            else
               print *, "external line from ", settings%Qmin, &
                    shower%partons(i)%p%ckkwscale
               if (parton_is_quark (shower%partons(i)%p%child2)) then
                  sf = SudakovQ (settings%Qmin, &
                       shower%partons(i)%p%ckkwscale, .true.)
                  print *, "SFQ = ", sf
               else if (parton_is_gluon (shower%partons(i)%p%child2)) then
                  sf = SudakovG (settings%Qmin, &
                       shower%partons(i)%p%ckkwscale, .true.)
                  print *, "SFG = ", sf
               else
                  print *, "not yet implemented (", &
                       shower%partons(i)%p%child2%type, ")"
                  sf = one
               end if
               weight = weight * min (one, sf)
            end if
         end if
      end do
end do

call tao_random_number(rand)

print *, "final weight: ", weight

!!!!!!! WRONG
veto = .false.
veto = (rand > weight)
if (veto) then
   return
end if

!!! finally perform the parton shower
!!! veto emissions that are too hard
```

```
      deallocate (scales)
    end subroutine ckkw_matching_apply
```

⟨*CKKW matching: procedures*⟩+≡
```
    function GammaQ (smallq, largeq, fsr) result (gamma)
      real(default), intent(in) :: smallq, largeq
      logical, intent(in) :: fsr
      real(default) :: gamma
      gamma = (8._default / three) / (pi * smallq)
      gamma = gamma * (log(largeq / smallq) - 0.75)
      if (fsr) then
         gamma = gamma * D_alpha_s_fsr (smallq**2)
      else
         gamma = gamma * D_alpha_s_isr (smallq**2)
      end if
    end function GammaQ
```

⟨*CKKW matching: procedures*⟩+≡
```
    function GammaG(smallq, largeq, fsr) result(gamma)
      real(default), intent(in) :: smallq, largeq
      logical, intent(in) :: fsr
      real(default) :: gamma
      gamma = 6._default / (pi * smallq)
      gamma = gamma *( log(largeq / smallq) - 11.0 / 12.0)
      if (fsr) then
         gamma = gamma * D_alpha_s_fsr(smallq**2)
      else
         gamma = gamma * D_alpha_s_isr(smallq**2)
      end if
    end function GammaG
```

⟨*CKKW matching: procedures*⟩+≡
```
    function GammaF (smallq, fsr) result (gamma)
      real(default), intent(in) :: smallq
      logical, intent(in) :: fsr
      real(default) :: gamma
      gamma = number_of_flavors (smallq) / (three * pi * smallq)
      if (fsr) then
         gamma = gamma * D_alpha_s_fsr (smallq**2)
      else
         gamma = gamma * D_alpha_s_isr (smallq**2)
      end if
    end function GammaF
```

⟨*CKKW matching: procedures*⟩+≡
```
    function SudakovQ (Q1, Q, fsr) result (sf)
      real(default), intent(in) :: Q1, Q
      logical, intent(in) :: fsr
      real(default) :: sf
      real(default) :: integral
      integer, parameter :: NTRIES = 100
      integer :: i
```

```
      real(default) :: rand
      integral = zero
      do i = 1, NTRIES
         call tao_random_number (rand)
         integral = integral + GammaQ (Q1 + rand * (Q - Q1), Q, fsr)
      end do
      integral = integral / NTRIES
      sf = exp (-integral)
   end function SudakovQ
```

⟨*CKKW matching: procedures*⟩+≡

```
   function SudakovG (Q1, Q, fsr) result (sf)
     real(default), intent(in) :: Q1, Q
     logical, intent(in) :: fsr
     real(default) :: sf
     real(default) :: integral
     integer, parameter :: NTRIES = 100
     integer :: i
     real(default) :: rand
     integral = zero
     do i = 1, NTRIES
        call tao_random_number (rand)
        integral = integral + GammaG (Q1 + rand * (Q - Q1), Q, fsr) + &
             GammaF (Q1 +rand * (Q - Q1), fsr)
     end do
     integral = integral / NTRIES
     sf = exp (-integral)
   end function SudakovG
```

CKKW (pseudo) weights

The type `ckkw_pseudo_shower_weights_t` gives the (relative) weights for different clusterings of the final particles, as given in Eq. (2.7) of hep-ph/0503281v1. Each particle has a binary labelling (power of 2) (first particle = 1, second particle = 2, third particle = 4, ...). Each recombination therefore corresponds to an integer, that is not a power of 2. Fur multiple subsequent recombinations, no different weights for different sequences of clustering are stored. It is assumed that the weight of a multiply recombined state is a combination of the states with one fewer recombination and that these states' contributions are proportional to their weights. For a $2->n$ event, the weights array thus has the size $2^{(2+n)-1}$. The `weights_by_type` array gives the weights depending on the type of the particle, the first index is the same as for weights, the second index gives the type of the new mother particle:

0: uncolored (γ, Z, W, Higgs)

1: colored (quark)

2: gluon

3: squark

4: gluino

alphaS gives the value for $alpha_s$ used in the generation of the matrix element. This is needed for the reweighting using the values for a running $alpha_s$ at the scales of the clusterings.

⟨ckkw_pseudo_weights.f90⟩≡

  ⟨*File header*⟩

```
  module ckkw_pseudo_weights
```

  ⟨*Use kinds*⟩
```
    use file_utils !NODEP!
    use constants !NODEP!
```

  ⟨*Standard module head*⟩

  ⟨*CKKW pseudo weights: public*⟩

  ⟨*CKKW pseudo weights: types*⟩

```
  contains
```

  ⟨*CKKW pseudo weights: procedures*⟩

```
  end module ckkw_pseudo_weights
```

⟨*CKKW pseudo weights: public*⟩≡
```
    public :: ckkw_pseudo_shower_weights_t
```

⟨*CKKW pseudo weights: types*⟩≡
```
    type :: ckkw_pseudo_shower_weights_t
       real(default) :: alphaS
       real(default), dimension(:), allocatable :: weights
       real(default), dimension(:,:), allocatable :: weights_by_type
    end type ckkw_pseudo_shower_weights_t
```

⟨*CKKW pseudo weights: public*⟩+≡
```
    public :: ckkw_pseudo_shower_weights_init
```

⟨*CKKW pseudo weights: procedures*⟩≡
```
    subroutine ckkw_pseudo_shower_weights_init (weights)
      type(ckkw_pseudo_shower_weights_t), intent(out) :: weights
      weights%alphaS = zero
      if (allocated (weights%weights))  deallocate(weights%weights)
      if (allocated (weights%weights_by_type))  &
           deallocate(weights%weights_by_type)
    end subroutine ckkw_pseudo_shower_weights_init
```

⟨*CKKW pseudo weights: public*⟩+≡
```
    public :: ckkw_pseudo_shower_weights_write
```

⟨*CKKW pseudo weights: procedures*⟩+≡
```
    subroutine ckkw_pseudo_shower_weights_write (weights, unit)
      type(ckkw_pseudo_shower_weights_t), intent(in) :: weights
      integer, intent(in), optional :: unit
      integer :: s, i, u
      u = output_unit (unit); if (u < 0) return
```

```
      s = size (weights%weights)
      write (u, "(1x,A)")  "CKKW (pseudo) shower weights: "
      do i = 1, s
         write (u, "(3x,I0,2(ES19.12))")  i, weights%weights(i), &
               weights%weights_by_type(i,:)
      end do
      write (u, "(3x,A,1x,I0)")  "alphaS =", weights%alphaS
    end subroutine ckkw_pseudo_shower_weights_write
```

This is a collection of dummy replacement routines.

⟨shower_dummy.f90⟩≡
```
  module pythia_dummy
    public :: pyinit
    public :: pygive
    public :: pylist
    public :: pyevnt
    public :: pyp
    public :: upinit
  contains
    subroutine pylist (i)
      integer, intent(in) :: i
      ⟨Shower dummy: error message⟩
    end subroutine pylist

    subroutine pyinit (frame, beam, target, win)
      character*(*), intent(in) ::  frame, beam, target
      double precision, intent(in) :: win
      ⟨Shower dummy: error message⟩
    end subroutine pyinit
    subroutine upinit
      ⟨Shower dummy: error message⟩
    end subroutine upinit
    subroutine pygive (chin)
      character chin*(*)
      ⟨Shower dummy: error message⟩
    end subroutine pygive
    subroutine pyevnt()
      ⟨Shower dummy: error message⟩
    end subroutine pyevnt
    subroutine pyexec()
      ⟨Shower dummy: error message⟩
    end subroutine pyexec
    function pyp(I,J)
      integer, intent(in) :: i,j
      double precision :: pyp
      ⟨Shower dummy: error message⟩
    end function pyp
  end module pythia_dummy


  module ckkw_pseudo_weights
    use kinds, only: default, double !NODEP!
```

```
   implicit none

   public :: ckkw_pseudo_shower_weights_t
   public :: ckkw_pseudo_shower_weights_write
   public :: ckkw_pseudo_shower_weights_init

   type :: ckkw_pseudo_shower_weights_t
      real(default) :: alphaS
      real(default), dimension(:), allocatable :: weights
      real(default), dimension(:,:), allocatable :: weights_by_type
   end type ckkw_pseudo_shower_weights_t

contains

   subroutine ckkw_pseudo_shower_weights_init (weights)
     type(ckkw_pseudo_shower_weights_t), intent(out) :: weights
     ⟨Shower dummy: error message⟩
   end subroutine ckkw_pseudo_shower_weights_init

   subroutine ckkw_pseudo_shower_weights_write (weights)
     type(ckkw_pseudo_shower_weights_t), intent(in) :: weights
     ⟨Shower dummy: error message⟩
   end subroutine ckkw_pseudo_shower_weights_write

end module ckkw_pseudo_weights


module shower_base
  use kinds, only: default !NODEP!
  use constants !NODEP!
  public :: shower_set_minenergy_timelike
  public :: shower_set_d_min_t
  public :: shower_set_d_nf
  public :: shower_set_d_running_alpha_s_fsr
  public :: shower_set_d_running_alpha_s_isr
  public :: shower_set_d_lambda_fsr
  public :: shower_set_d_lambda_isr
  public :: shower_set_d_constantalpha_s
  public :: shower_set_maxz_isr
  public :: shower_set_isr_pt_ordered
  public :: shower_set_isr_angular_ordered
  public :: shower_set_primordial_kt_width
  public :: shower_set_primordial_kt_cutoff
  public :: shower_set_tscalefactor_isr
  public :: shower_set_isr_only_onshell_emitted_partons
  public :: shower_set_pdf_func
  public :: shower_set_pdf_set
  public :: shower_pdf

  real(default), public :: D_Min_t = one

  interface
     subroutine shower_pdf (set, x, q, ff)
       integer, intent(in) :: set
```

```
        double precision, intent(in) :: x, q
        double precision, dimension(-6:6), intent(out) :: ff
      end subroutine shower_pdf
  end interface
contains
  subroutine shower_set_minenergy_timelike (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_minenergy_timelike
  subroutine shower_set_d_min_t (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_min_t
  subroutine shower_set_d_nf (input)
    integer, intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_nf
  subroutine shower_set_isr_pt_ordered (input)
    logical, intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_isr_pt_ordered
  subroutine shower_set_isr_angular_ordered (input)
    logical, intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_isr_angular_ordered
  subroutine shower_set_d_lambda_fsr (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_lambda_fsr
  subroutine shower_set_d_lambda_isr (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_lambda_isr
  subroutine shower_set_d_running_alpha_s_fsr (input)
    logical, intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_running_alpha_s_fsr
  subroutine shower_set_d_running_alpha_s_isr (input)
    logical, intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_running_alpha_s_isr
  subroutine shower_set_d_constantalpha_s (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_d_constantalpha_s
  subroutine shower_set_maxz_isr (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_maxz_isr
  subroutine shower_set_primordial_kt_width (input)
    real(default), intent(in) :: input
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_primordial_kt_width
  subroutine shower_set_primordial_kt_cutoff (input)
```

```
      real(default), intent(in) :: input
      ⟨Shower dummy: error message⟩
    end subroutine shower_set_primordial_kt_cutoff
    subroutine shower_set_tscalefactor_isr (input)
      real(default), intent(in) :: input
      ⟨Shower dummy: error message⟩
    end subroutine shower_set_tscalefactor_isr
    subroutine shower_set_isr_only_onshell_emitted_partons (input)
      logical, intent(in) :: input
      ⟨Shower dummy: error message⟩
    end subroutine shower_set_isr_only_onshell_emitted_partons
    subroutine shower_set_pdf_func(func)
      procedure(shower_pdf), pointer, intent(in) :: func
      ⟨Shower dummy: error message⟩
    end subroutine shower_set_pdf_func

    subroutine shower_set_pdf_set(set)
      integer, intent(in) :: set
      ⟨Shower dummy: error message⟩
    end subroutine shower_set_pdf_set
end module shower_base

module shower_partons
  use kinds, only: default !NODEP!
  use constants !NODEP!
  use lorentz !NODEP!

⟨Shower partons: public⟩

⟨Shower partons: types⟩

contains
  subroutine parton_set_simulated (prt, sim)
    type(parton_t), intent(inout) :: prt
    logical, intent(in), optional :: sim
    ⟨Shower dummy: error message⟩
  end subroutine parton_set_simulated
  subroutine parton_set_momentum (prt, EE, ppx, ppy, ppz)
    type(parton_t), intent(inout) :: prt
    real(default), intent(in) :: EE, ppx, ppy, ppz
    ⟨Shower dummy: error message⟩
  end subroutine parton_set_momentum
  subroutine parton_set_initial (prt, initial)
    type(parton_t), intent(inout) :: prt
    type(parton_t), intent(in) , target :: initial
    ⟨Shower dummy: error message⟩
  end subroutine parton_set_initial
end module shower_partons

module shower_core
  use kinds, only: default
  use shower_base
  use shower_partons
  use ckkw_pseudo_weights
```

```
  use pythia_dummy
```

⟨*Shower core: public*⟩

⟨*Shower core: types*⟩

⟨*Shower core: parameters*⟩

```
contains
    function shower_get_next_free_nr (shower) result(next_number)
      class(shower_t), intent(inout) :: shower
      integer :: next_number
```
      ⟨*Shower dummy: error message*⟩
```
    end function shower_get_next_free_nr
    function shower_generate_next_isr_branching (shower) result (next_brancher)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t) :: next_brancher
```
      ⟨*Shower dummy: error message*⟩
```
    end function shower_generate_next_isr_branching
    function shower_generate_next_isr_branching_veto (shower) &
         result (next_brancher)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t) :: next_brancher
```
      ⟨*Shower dummy: error message*⟩
```
    end function shower_generate_next_isr_branching_veto
    subroutine shower_generate_fsr_for_partons_emitted_in_isr (shower)
      class(shower_t), intent(inout) :: shower
```
      ⟨*Shower dummy: error message*⟩
```
    end subroutine shower_generate_fsr_for_partons_emitted_in_isr
    subroutine interaction_generate_primordial_kt (interaction)
      type(shower_interaction_t), intent(inout) :: interaction
```
      ⟨*Shower dummy: error message*⟩
```
    end subroutine interaction_generate_primordial_kt
    subroutine shower_generate_primordial_kt (shower)
      class(shower_t), intent(inout) :: shower
```
      ⟨*Shower dummy: error message*⟩
```
    end subroutine shower_generate_primordial_kt
    subroutine shower_interaction_generate_fsr_2ton (shower, interaction)
      class(shower_t), intent(inout) :: shower
      type(shower_interaction_t), intent(inout) :: interaction
```
      ⟨*Shower dummy: error message*⟩
```
    end subroutine shower_interaction_generate_fsr_2ton
    subroutine shower_execute_next_isr_branching (shower, prtp)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t), intent(inout) :: prtp
```
      ⟨*Shower dummy: error message*⟩
```
    end subroutine shower_execute_next_isr_branching
    subroutine shower_boost_to_labframe (shower)
      class(shower_t), intent(inout) :: shower
```
      ⟨*Shower dummy: error message*⟩
```
    end subroutine shower_boost_to_labframe
    subroutine shower_get_final_colored_ME_partons(shower, partons)
      class(shower_t), intent(in) :: shower
      type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: &
```

```
        partons
    ⟨Shower dummy: error message⟩
  end subroutine shower_get_final_colored_ME_partons
  subroutine shower_get_final_partons (shower, partons, include_remnants)
    type(shower_t), intent(in) :: shower
    type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: partons
    logical, intent(in), optional :: include_remnants
    ⟨Shower dummy: error message⟩
  end subroutine shower_get_final_partons
  subroutine shower_write (shower)
    class(shower_t), intent(inout) :: shower
    ⟨Shower dummy: error message⟩
  end subroutine shower_write
  subroutine shower_sort_partons (shower)
    class(shower_t), intent(inout) :: shower
    ⟨Shower dummy: error message⟩
  end subroutine shower_sort_partons
  subroutine shower_add_interaction_2ton (shower, partons)
    class(shower_t), intent(inout) :: shower
    type(parton_pointer_t), intent(inout), dimension(:), allocatable :: &
        partons
    ⟨Shower dummy: error message⟩
  end subroutine shower_add_interaction_2ton
  subroutine shower_add_interaction_2ton_CKKW &
      (shower, partons, ckkw_pseudo_weights)
    class(shower_t), intent(inout) :: shower
    type(parton_pointer_t), intent(in), dimension(:), allocatable :: partons
    type(ckkw_pseudo_shower_weights_t), intent(in) :: ckkw_pseudo_weights
    ⟨Shower dummy: error message⟩
  end subroutine shower_add_interaction_2ton_CKKW
  subroutine shower_simulate_no_isr_shower (shower)
    class(shower_t), intent(inout) :: shower
    ⟨Shower dummy: error message⟩
  end subroutine shower_simulate_no_isr_shower
  subroutine shower_simulate_no_fsr_shower (shower)
    class(shower_t), intent(inout) :: shower
    ⟨Shower dummy: error message⟩
  end subroutine shower_simulate_no_fsr_shower
  subroutine shower_update_beamremnants (shower)
    class(shower_t), intent(in) :: shower
    ⟨Shower dummy: error message⟩
  end subroutine shower_update_beamremnants
  subroutine shower_set_next_color_nr (shower, index)
    class(shower_t), intent(in) :: shower
    integer, intent(in) :: index
    ⟨Shower dummy: error message⟩
  end subroutine shower_set_next_color_nr
  subroutine shower_create (shower)
    class(shower_t), intent(inout) :: shower
    ⟨Shower dummy: error message⟩
  end subroutine shower_create
  subroutine shower_final (shower)
    class(shower_t), intent(inout) :: shower
    ⟨Shower dummy: error message⟩
```

```
      end subroutine shower_final
      subroutine shower_write_lhef (shower, unit)
        class(shower_t), intent(in) :: shower
        integer, intent(in), optional :: unit
        ⟨Shower dummy: error message⟩
      end subroutine shower_write_lhef
      function shower_interaction_get_s (interaction) result(s)
        type(shower_interaction_t), intent(in) :: interaction
        real(default) :: s
        ⟨Shower dummy: error message⟩
      end function shower_interaction_get_s
      function shower_get_ISR_scale (shower) result (scale)
        class(shower_t), intent(in) :: shower
        real(default) :: scale
        ⟨Shower dummy: error message⟩
      end function shower_get_ISR_scale
      function shower_get_next_color_nr (shower) result(next_color)
        class(shower_t), intent(inout) :: shower
        integer :: next_color
        ⟨Shower dummy: error message⟩
      end function shower_get_next_color_nr
      subroutine shower_set_max_isr_scale (shower, newscale)
        class(shower_t), intent(inout) :: shower
        real(default), intent(in) :: newscale
        ⟨Shower dummy: error message⟩
      end subroutine shower_set_max_isr_scale
      subroutine shower_add_child (shower, prt, child)
        class(shower_t), intent(inout) :: shower
        type(parton_t), intent(in) :: prt
        integer, intent(in) :: child
        ⟨Shower dummy: error message⟩
      end subroutine shower_add_child
      subroutine shower_add_parent (shower, prt)
        class(shower_t), intent(inout) :: shower
        type(parton_t), intent(in) :: prt
        ⟨Shower dummy: error message⟩
      end subroutine shower_add_parent
end module shower_core

module shower_topythia
⟨Use kinds⟩
  use shower_base
  use shower_partons
  use shower_core
⟨Shower2pythia: public⟩
contains
  subroutine shower_converttopythia (shower)
    type(shower_t), intent(in) :: shower
    ⟨Shower dummy: error message⟩
    end subroutine shower_converttopythia
end module shower_topythia

module mlm_matching
⟨Use kinds⟩
```

```
    use kinds, only: double !NODEP!
    use constants !NODEP!
    use lorentz !NODEP!
```

⟨*MLM matching: public*⟩

⟨*MLM matching: types*⟩

```
contains

  subroutine mlm_matching_settings_write (settings, unit)
    type(mlm_matching_settings_t), intent(in) :: settings
    integer, intent(in), optional :: unit
    ⟨Shower dummy: error message⟩
  end subroutine mlm_matching_settings_write

  subroutine mlm_matching_data_final (data)
    type(mlm_matching_data_t), intent(inout) :: data
    ⟨Shower dummy: error message⟩
  end subroutine mlm_matching_data_final

  subroutine mlm_matching_apply (data, settings, vetoed)
    type(mlm_matching_data_t), intent(inout) :: data
    type(mlm_matching_settings_t), intent(in) :: settings
    logical, intent(out) :: vetoed
    ⟨Shower dummy: error message⟩
  end subroutine mlm_matching_apply
end module mlm_matching



module ckkw_matching
```
⟨*Use kinds*⟩
```
  use kinds, only: double !NODEP!
  use shower_core
  use ckkw_pseudo_weights
```

⟨*Standard module head*⟩

⟨*CKKW matching: public*⟩

⟨*CKKW matching: types*⟩

```
contains

  subroutine ckkw_matching_apply (shower, settings, weights, veto)
    type(shower_t), intent(inout) :: shower
    type(ckkw_matching_settings_t), intent(in) :: settings
    type(ckkw_pseudo_shower_weights_t), intent(in) :: weights
    logical, intent(out) :: veto
    veto = .false.
    ⟨Shower dummy: error message⟩
  end subroutine ckkw_matching_apply
```

```
      end module ckkw_matching
```

⟨*Shower dummy: error message*⟩≡

```
      write (0, "(A)")  "****************************************************************"
      write (0, "(A)")  "*** Error: Shower has not been enabled, WHIZARD terminates ***"
      write (0, "(A)")  "****************************************************************"
      stop
```

Chapter 17

Events

With a process object and the associated methods at hand, we can generate events for elementary processes and, by subsequent transformation, for complete physical processes.

We have the following modules:

event_transforms Abstract base type for transforming a physical process with process instance and included evaluators, etc., into a new object. The following modules extend this base type.

decays Combine the elementary process with elementary decay processes and thus transform the elementary event into a decayed event, still at the parton level.

showers (not re-activated yet) Create QED/QCD showers out of the partons that are emitted by elementary processes. This should be interleaved with showering of radiated particles (structure functions) and multiple interactions.

hadrons (not implemented yet) Apply hadronization to the partonic events, interleaved with hadron decays. (The current setup relies on hadronizing partonic events externally.)

events Combine all pieces to generate full events.

17.1 Abstract Event Transforms

⟨`event_transforms.f90`⟩≡

  ⟨*File header*⟩

  `module event_transforms`

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    `use diagnostics !NODEP!`
    `use unit_tests`
    `use os_interface`

```
    use sm_qcd
    use models
    use state_matrices
    use interactions
    use particles
    use subevents
    use process_libraries
    use prc_core
    use prc_test
    use rng_base
    use mci_base
    use mci_midpoint
    use phs_base
    use phs_single
    use processes
    use process_stacks
```

⟨*Standard module head*⟩

⟨*Event transforms: public*⟩

⟨*Event transforms: types*⟩

⟨*Event transforms: interfaces*⟩

```
contains
```

⟨*Event transforms: procedures*⟩

⟨*Event transforms: tests*⟩

```
end module event_transforms
```

17.1.1   Abstract base type

Essentially, all methods are abstract, but some get minimal base versions. We know that there will be a random-number generator at top level, and that we will relate to an elementary process.

The model is stored separately. It may contain modified setting that differ from the model instance stored in the process object.

Each event transform contains a particle set that it can fill for further use. There is a flag the indicates this.

We will collect event transforms in a list, therefore we include a `previous` and `next` pointers.

⟨*Event transforms: public*⟩≡
```
    public :: evt_t
```

⟨*Event transforms: types*⟩≡
```
    type, abstract :: evt_t
       type(process_t), pointer :: process => null ()
       type(process_instance_t), pointer :: process_instance => null ()
       type(model_t), pointer :: model => null ()
```

```
      class(rng_t), allocatable :: rng
      integer :: rejection_count = 0
      logical :: particle_set_exists = .false.
      type(particle_set_t) :: particle_set
      class(evt_t), pointer :: previous => null ()
      class(evt_t), pointer :: next => null ()
   contains
   ⟨Event transforms: evt: TBP⟩
   end type evt_t
```

Finalizer. In any case, we finalize the r.n.g. The process instance is a pointer and should not be finalized here.

⟨*Event transforms: evt: TBP*⟩≡
```
    procedure :: final => evt_final
    procedure :: base_final => evt_final
```

⟨*Event transforms: procedures*⟩≡
```
    subroutine evt_final (object)
      class(evt_t), intent(inout) :: object
      if (allocated (object%rng))  call object%rng%final ()
      if (object%particle_set_exists) &
           call particle_set_final (object%particle_set)
    end subroutine evt_final
```

Output. We can print r.n.g. info.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure :: base_write => evt_write
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_write (object, unit, verbose, testflag)
      class(evt_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, testflag
      integer :: u
      u = output_unit (unit)
      if (associated (object%process)) then
         write (u, "(3x,A,A,A)")   "Associated process: '", &
              char (object%process%get_id ()), "'"
      end if
      if (allocated (object%rng)) then
         call object%rng%write (u, 1)
         write (u, "(3x,A,I0)")   "Number of tries = ", object%rejection_count
      end if
      if (object%particle_set_exists) then
         call write_separator (u)
         call particle_set_write (object%particle_set, u, testflag)
      end if
    end subroutine evt_write
```

Connect the transform with a process instance (and thus with the associated process). Use this to allocate the master random-number generator.

This is not an initializer; we may initialize the transform by implementation-specific methods.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure :: connect => evt_connect
    procedure :: base_connect => evt_connect
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_connect (evt, process_instance, model, process_stack)
      class(evt_t), intent(inout), target :: evt
      type(process_instance_t), intent(in), target :: process_instance
      type(model_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      evt%process => process_instance%process
      evt%process_instance => process_instance
      evt%model => model
      call evt%process%make_rng (evt%rng)
    end subroutine evt_connect
```

Reset internal state.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure :: reset => evt_reset
    procedure :: base_reset => evt_reset
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_reset (evt)
      class(evt_t), intent(inout) :: evt
      evt%rejection_count = 0
      evt%particle_set_exists = .false.
    end subroutine evt_reset
```

Prepare for a new event: reset internal state, if necessary. We provide MCI and term index of the parent process.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure (evt_prepare_new_event), deferred :: prepare_new_event
```

⟨*Event transforms: interfaces*⟩≡
```
    interface
       subroutine evt_prepare_new_event (evt, i_mci, i_term)
         import
         class(evt_t), intent(inout) :: evt
         integer, intent(in) :: i_mci, i_term
       end subroutine evt_prepare_new_event
    end interface
```

Generate a weighted event, using a valid initiator event in the process instance, and the random-number generator. The returned event probability should be a number between zero and one that we can use for rejection.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure (evt_generate_weighted), deferred :: generate_weighted
```

⟨*Event transforms: interfaces*⟩+≡
```
    abstract interface
       subroutine evt_generate_weighted (evt, probability)
```

```
        import
        class(evt_t), intent(inout) :: evt
        real(default), intent(out) :: probability
      end subroutine evt_generate_weighted
    end interface
```

The unweighted event generation routine is actually implemented. It uses the random-number generator for simple rejection. Of course, the implementation may override this and implement a different way of generating an unweighted event.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure :: generate_unweighted => evt_generate_unweighted
    procedure :: base_generate_unweighted => evt_generate_unweighted
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_generate_unweighted (evt)
      class(evt_t), intent(inout) :: evt
      real(default) :: p, x
      evt%rejection_count = 0
      REJECTION: do
         evt%rejection_count = evt%rejection_count + 1
         call evt%generate_weighted (p)
         if (signal_is_pending ())  return
         call evt%rng%generate (x)
         if (x < p)  exit REJECTION
      end do REJECTION
    end subroutine evt_generate_unweighted
```

Make a particle set. This should take the most recent evaluator (or whatever stores the event), factorize the density matrix if necessary, and store as a particle set.

If applicable, the factorization should make use of the `factorization_mode` and `keep_correlations` settings.

The values **r**, if set, should control the factorization in more detail, e.g., bypassing the random-number generator.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure (evt_make_particle_set), deferred :: make_particle_set
```

⟨*Event transforms: interfaces*⟩+≡
```
    interface
      subroutine evt_make_particle_set &
           (evt, factorization_mode, keep_correlations, r)
        import
        class(evt_t), intent(inout) :: evt
        integer, intent(in) :: factorization_mode
        logical, intent(in) :: keep_correlations
        real(default), dimension(:), intent(in), optional :: r
      end subroutine evt_make_particle_set
    end interface
```

Copy an existing particle set into the event record. This bypasses all methods to evaluate the internal state, but may be sufficient for further processing.

⟨*Event transforms: evt: TBP*⟩+≡

```
        procedure :: set_particle_set => evt_set_particle_set
```
⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_set_particle_set (evt, particle_set, i_mci, i_term)
      class(evt_t), intent(inout) :: evt
      type(particle_set_t), intent(in) :: particle_set
      integer, intent(in) :: i_term, i_mci
      call evt%prepare_new_event (i_mci, i_term)
      evt%particle_set = particle_set
      evt%particle_set_exists = .true.
    end subroutine evt_set_particle_set
```

This procedure can help in the previous task, if the particles are available in
the form of an interaction object. (We need two interactions, one with color
summed over, and one with the probability distributed among flows.)

  We use the two values from the random number generator for factorizing the
state. For testing purposes, we can provide those numbers explicitly.

⟨*Event transforms: evt: TBP*⟩+≡
```
    procedure :: factorize_interactions => evt_factorize_interactions
```
⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_factorize_interactions &
        (evt, int_matrix, int_flows, factorization_mode, keep_correlations, r)
      class(evt_t), intent(inout) :: evt
      type(interaction_t), intent(in), target :: int_matrix, int_flows
      integer, intent(in) :: factorization_mode
      logical, intent(in) :: keep_correlations
      real(default), dimension(:), intent(in), optional :: r
      real(default), dimension(2) :: x
      if (present (r)) then
         if (size (r) == 2) then
            x = r
         else
            call msg_bug ("event factorization: size of r array must be 2")
         end if
      else
         call evt%rng%generate (x)
      end if
      call particle_set_init (evt%particle_set, evt%particle_set_exists, &
           int_matrix, int_flows, factorization_mode, x, &
           keep_correlations, keep_virtual=.true.)
      evt%particle_set_exists = .true.
    end subroutine evt_factorize_interactions
```

Mark the incoming particles as incoming in the particle set. This is necessary
because in the interaction objects they are usually marked as virtual.

  In the inquiry functions we set the term index to one; the indices of beams
and incoming particles should be identical for all process terms.

  We use the initial elementary process for obtaining the indices. Thus, we
implicitly assume that the beam and incoming indices stay the same across event
transforms. If this is not true for a transform (say, MPI), it should override this
method.

⟨*Event transforms: evt: TBP*⟩+≡

```
    procedure :: tag_incoming => evt_tag_incoming
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_tag_incoming (evt)
      class(evt_t), intent(inout) :: evt
      integer :: i_term, n_in
      integer, dimension(:), allocatable :: beam_index, in_index
      n_in = evt%process%get_n_in ()
      i_term = 1
      allocate (beam_index (n_in))
      call evt%process_instance%get_beam_index (i_term, beam_index)
      call particle_set_reset_status (evt%particle_set, &
           beam_index, PRT_BEAM)
      allocate (in_index (n_in))
      call evt%process_instance%get_in_index (i_term, in_index)
      call particle_set_reset_status (evt%particle_set, &
           in_index, PRT_INCOMING)
    end subroutine evt_tag_incoming
```

17.1.2 Implementation: Trivial transform

This transform contains just a pointer to process and process instance. The
generate methods do nothing.

⟨*Event transforms: public*⟩+≡
```
    public :: evt_trivial_t
```

⟨*Event transforms: types*⟩+≡
```
    type, extends (evt_t) :: evt_trivial_t
     contains
     ⟨Event transforms: evt trivial: TBP⟩
    end type evt_trivial_t
```

The finalizer is trivial. Some output:

⟨*Event transforms: evt trivial: TBP*⟩≡
```
    procedure :: write => evt_trivial_write
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_trivial_write (object, unit, verbose, testflag)
      class(evt_trivial_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose, testflag
      integer :: u
      u = output_unit (unit)
      call write_separator_double (u)
      write (u, "(1x,A)")  "Event transform: trivial (hard process)"
      call write_separator (u)
      call object%base_write (u, testflag = testflag)
    end subroutine evt_trivial_write
```

Nothing to do here:

⟨*Event transforms: evt trivial: TBP*⟩+≡
```
    procedure :: prepare_new_event => evt_trivial_prepare_new_event
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_trivial_prepare_new_event (evt, i_mci, i_term)
      class(evt_trivial_t), intent(inout) :: evt
      integer, intent(in) :: i_mci, i_term
      call evt%reset ()
    end subroutine evt_trivial_prepare_new_event
```

The weighted generator is, surprisingly, trivial.

⟨*Event transforms: evt trivial: TBP*⟩+≡
```
    procedure :: generate_weighted => evt_trivial_generate_weighted
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_trivial_generate_weighted (evt, probability)
      class(evt_trivial_t), intent(inout) :: evt
      real(default), intent(out) :: probability
      probability = 1
    end subroutine evt_trivial_generate_weighted
```

This routine makes a particle set, using the associated process instance as-is.

⟨*Event transforms: evt trivial: TBP*⟩+≡
```
    procedure :: make_particle_set => evt_trivial_make_particle_set
```

⟨*Event transforms: procedures*⟩+≡
```
    subroutine evt_trivial_make_particle_set &
        (evt, factorization_mode, keep_correlations, r)
      class(evt_trivial_t), intent(inout) :: evt
      integer, intent(in) :: factorization_mode
      logical, intent(in) :: keep_correlations
      real(default), dimension(:), intent(in), optional :: r
      integer :: i_term
      type(interaction_t), pointer :: int_matrix, int_flows
      if (evt%process_instance%is_complete_event ()) then
         call evt%process_instance%select_i_term (i_term)
         int_matrix => evt%process_instance%get_matrix_int_ptr (i_term)
         int_flows  => evt%process_instance%get_flows_int_ptr (i_term)
         call evt%factorize_interactions (int_matrix, int_flows, &
             factorization_mode, keep_correlations, r)
         call evt%tag_incoming ()
      else
         call msg_bug ("Event factorization: event is incomplete")
      end if
    end subroutine evt_trivial_make_particle_set
```

17.1.3 Unit tests

⟨*Event transforms: public*⟩+≡
```
    public :: event_transforms_test
```

⟨*Event transforms: tests*⟩≡
```
    subroutine event_transforms_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Event transforms: execute tests*⟩

```
          end subroutine event_transforms_test
```

Test trivial event transform

The trivial transform, as an instance of the abstract transform, does nothing
but to trigger event generation for an elementary process.

⟨*Event transforms: execute tests*⟩≡
```
    call test (event_transforms_1, "event_transforms_1", &
         "trivial event transform", &
         u, results)
```

⟨*Event transforms: tests*⟩+≡
```
    subroutine event_transforms_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_library_t), target :: lib
      type(string_t) :: libname, procname1, run_id
      class(prc_core_t), allocatable :: core_template
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
      class(evt_t), allocatable :: evt
      integer :: factorization_mode
      logical :: keep_correlations

      write (u, "(A)")  "* Test output: event_transforms_1"
      write (u, "(A)")  "*   Purpose: handle trivial transform"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize environment and parent process"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call model_list%read_model (var_str ("Test"), &
           var_str ("Test.mdl"), os_data, model)

      libname = "event_transforms_1_lib"
      procname1 = "event_transforms_1_p"
      run_id = "event_transforms_1"

      call prc_test_create_library (libname, lib, &
           scattering = .true., procname1 = procname1)
      call reset_interaction_counter ()
      allocate (process)
      call process%init &
```

```
      (procname1, run_id, lib, os_data, qcd, rng_factory, model_list)

allocate (test_t :: core_template)
allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
     (1, core_template, mci_template, phs_config_template)

sqrts = 1000
call process%setup_beams_sqrts (sqrts)
call process%configure_phs ()
call process%setup_mci ()
call process%setup_terms ()

allocate (process_instance)
call process_instance%init (process)
call process%integrate (process_instance, 1, n_it=1, n_calls=100)
call process%final_integration (1)
call process_instance%final ()
deallocate (process_instance)

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
call process_instance%init_simulation (1)

write (u, "(A)")  "* Initialize trivial event transform"
write (u, "(A)")

allocate (evt_trivial_t :: evt)
call evt%connect (process_instance, model)

write (u, "(A)")  "* Generate event and subsequent transform"
write (u, "(A)")

call process%generate_unweighted_event (process_instance, 1)
call process_instance%evaluate_event_data ()

call evt%prepare_new_event (1, 1)
call evt%generate_unweighted ()

select type (evt)
type is (evt_trivial_t)
   call write_separator_double (u)
   call evt%write (u)
   call write_separator_double (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Obtain particle set"
write (u, "(A)")

factorization_mode = FM_IGNORE_HELICITY
keep_correlations = .false.
```

```
      call evt%make_particle_set (factorization_mode, keep_correlations)

      select type (evt)
      type is (evt_trivial_t)
         call write_separator_double (u)
         call evt%write (u)
         call write_separator_double (u)
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call evt%final ()
      call process_instance%final ()
      deallocate (process_instance)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: event_transforms_1"

    end subroutine event_transforms_1
```

17.2 Decays

⟨decays.f90⟩≡
  ⟨*File header*⟩

```
  module decays
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use unit_tests
    use os_interface

    use sm_qcd
    use flavors
    use quantum_numbers
    use state_matrices
    use interactions
    use evaluators
    use variables
    use models
    use rng_base
    use selectors
    use prc_core
    use prc_test
    use process_libraries
```

```
    use mci_base
    use mci_midpoint
    use phs_base
    use phs_single
    use parton_states
    use processes
    use process_stacks
    use event_transforms
```

⟨*Standard module head*⟩

⟨*Decays: public*⟩

⟨*Decays: types*⟩

⟨*Decays: interfaces*⟩

```
contains
```

⟨*Decays: procedures*⟩

⟨*Decays: tests*⟩

```
end module decays
```

17.2.1 Final-State Particle Configuration

A final-state particle may be either stable or unstable. Here is an empty abstract type as the parent of both, with holds just the flavor information.

⟨*Decays: types*⟩≡
```
    type, abstract :: any_config_t
     contains
     ⟨Decays: any config: TBP⟩
    end type any_config_t
```

Finalizer, depends on the implementation.

⟨*Decays: any config: TBP*⟩≡
```
    procedure (any_config_final), deferred :: final
```

⟨*Decays: interfaces*⟩≡
```
    interface
       subroutine any_config_final (object)
         import
         class(any_config_t), intent(inout) :: object
       end subroutine any_config_final
    end interface
```

The output is also deferred:

⟨*Decays: any config: TBP*⟩+≡
```
    procedure (any_config_write), deferred :: write
```

⟨*Decays: interfaces*⟩+≡
```
interface
   subroutine any_config_write (object, unit, indent, verbose)
     import
     class(any_config_t), intent(in) :: object
     integer, intent(in), optional :: unit, indent
     logical, intent(in), optional :: verbose
   end subroutine any_config_write
end interface
```

This is a container for a stable or unstable particle configurator. We need this wrapper for preparing arrays that mix stable and unstable particles.

⟨*Decays: types*⟩+≡
```
type :: particle_config_t
   class(any_config_t), allocatable :: c
end type particle_config_t
```

17.2.2   Final-State Particle

In theory, for the particle instance we only need to consider the unstable case. However, it is more straightforward to treat configuration and instance on the same footing, and to introduce a wrapper for particle objects as above. This also works around a compiler bug in gfortran.

⟨*Decays: types*⟩+≡
```
type, abstract :: any_t
 contains
 ⟨Decays: any: TBP⟩
end type any_t
```

Finalizer, depends on the implementation.

⟨*Decays: any: TBP*⟩≡
```
procedure (any_final), deferred :: final
```

⟨*Decays: interfaces*⟩+≡
```
interface
   subroutine any_final (object)
     import
     class(any_t), intent(inout) :: object
   end subroutine any_final
end interface
```

The output is also deferred:

⟨*Decays: any: TBP*⟩+≡
```
procedure (any_write), deferred :: write
```

⟨*Decays: interfaces*⟩+≡
```
interface
   subroutine any_write (object, unit, indent)
     import
     class(any_t), intent(in) :: object
     integer, intent(in), optional :: unit, indent
```

```
          end subroutine any_write
      end interface
```

This is a container for a stable or unstable outgoing particle. We need this wrapper for preparing arrays that mix stable and unstable particles.

⟨*Decays: types*⟩+≡
```
    type :: particle_out_t
       class(any_t), allocatable :: c
    end type particle_out_t
```

17.2.3 Decay Term Configuration

A decay term is a distinct final state, corresponding to a process term. Each decay process may give rise to several terms with, possibly, differing flavor content.

⟨*Decays: types*⟩+≡
```
    type :: decay_term_config_t
       type(particle_config_t), dimension(:), allocatable :: prt
     contains
     ⟨Decays: decay term config: TBP⟩
    end type decay_term_config_t
```

Finalizer, recursive.

⟨*Decays: decay term config: TBP*⟩≡
```
    procedure :: final => decay_term_config_final
```

⟨*Decays: procedures*⟩≡
```
    recursive subroutine decay_term_config_final (object)
      class(decay_term_config_t), intent(inout) :: object
      integer :: i
      if (allocated (object%prt)) then
         do i = 1, size (object%prt)
            if (allocated (object%prt(i)%c))  call object%prt(i)%c%final ()
         end do
      end if
    end subroutine decay_term_config_final
```

Output, with optional indentation

⟨*Decays: decay term config: TBP*⟩+≡
```
    procedure :: write => decay_term_config_write
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_term_config_write (object, unit, indent, verbose)
      class(decay_term_config_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      logical, intent(in), optional :: verbose
      integer :: i, j, u, ind
      logical :: verb
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      verb = .true.;  if (present (verbose))  verb = verbose
```

```
      call write_indent (u, ind)
      write (u, "(1x,A)", advance="no")  "Final state:"
      do i = 1, size (object%prt)
         select type (prt_config => object%prt(i)%c)
         type is (stable_config_t)
            write (u, "(1x,A)", advance="no") &
                char (flavor_get_name (prt_config%flv(1)))
            do j = 2, size (prt_config%flv)
               write (u, "(':',A)", advance="no") &
                   char (flavor_get_name (prt_config%flv(j)))
            end do
         type is (unstable_config_t)
            write (u, "(1x,A)", advance="no") &
                char (flavor_get_name (prt_config%flv))
         end select
      end do
      write (u, *)
      if (verb) then
         do i = 1, size (object%prt)
            call object%prt(i)%c%write (u, ind)
         end do
      end if
   end subroutine decay_term_config_write
```

Initialize, given a set of flavors. For each flavor, we must indicate whether the
particle is stable. The second index of the flavor array runs over alternatives for
each decay product; alternatives are allowed only if the decay product is itself
stable.

⟨Decays: decay term config: TBP⟩+≡
```
   procedure :: init => decay_term_config_init
```

⟨Decays: procedures⟩+≡
```
   recursive subroutine decay_term_config_init &
       (term, flv, stable, model, process_stack)
     class(decay_term_config_t), intent(out) :: term
     type(flavor_t), dimension(:,:), intent(in) :: flv
     logical, dimension(:), intent(in) :: stable
     type(model_t), intent(in), target :: model
     type(process_stack_t), intent(in), optional :: process_stack
     type(string_t), dimension(:), allocatable :: decay
     integer :: i
     allocate (term%prt (size (flv, 1)))
     do i = 1, size (flv, 1)
        associate (prt => term%prt(i))
          if (stable(i)) then
             allocate (stable_config_t :: prt%c)
          else
             allocate (unstable_config_t :: prt%c)
          end if
          select type (prt_config => prt%c)
          type is (stable_config_t)
             call prt_config%init (flv(i,:))
          type is (unstable_config_t)
             if (all (flv(i,:) == flv(i,1))) then
```

```
                call prt_config%init (flv(i,1))
                call flavor_get_decays (flv(i,1), decay)
                call prt_config%init_decays (decay, model, process_stack)
            else
                call prt_config%write ()
                call msg_fatal ("Decay configuration: &
                    &unstable product must be unique")
            end if
        end select
    end associate
  end do
end subroutine decay_term_config_init
```

Recursively compute widths and branching ratios for all unstable particles.

⟨*Decays: decay term config: TBP*⟩+≡
```
    procedure :: compute => decay_term_config_compute
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_term_config_compute (term)
      class(decay_term_config_t), intent(inout) :: term
      integer :: i
      do i = 1, size (term%prt)
         select type (unstable_config => term%prt(i)%c)
         type is (unstable_config_t)
            call unstable_config%compute ()
         end select
      end do
    end subroutine decay_term_config_compute
```

17.2.4   Decay Term

A decay term instance is selected when we generate an event for the associated
process instance. When evaluated, it triggers further decays down the chain.

Only unstable products are allocated as child particles.

⟨*Decays: types*⟩+≡
```
    type :: decay_term_t
       type(decay_term_config_t), pointer :: config => null ()
       type(particle_out_t), dimension(:), allocatable :: particle_out
     contains
     ⟨Decays: decay term: TBP⟩
    end type decay_term_t
```

Finalizer.

⟨*Decays: decay term: TBP*⟩≡
```
    procedure :: final => decay_term_final
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_term_final (object)
      class(decay_term_t), intent(inout) :: object
      integer :: i
      if (allocated (object%particle_out)) then
         do i = 1, size (object%particle_out)
```

2332

```
            call object%particle_out(i)%c%final ()
         end do
      end if
   end subroutine decay_term_final
```

Output.

⟨*Decays: decay term: TBP*⟩+≡
```
   procedure :: write => decay_term_write
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine decay_term_write (object, unit, indent)
     class(decay_term_t), intent(in) :: object
     integer, intent(in), optional :: unit, indent
     integer :: i, u, ind
     u = output_unit (unit)
     ind = 0;  if (present (indent))  ind = indent
     call object%config%write (u, ind, verbose = .false.)
     do i = 1, size (object%particle_out)
        call object%particle_out(i)%c%write (u, ind)
     end do
   end subroutine decay_term_write
```

Recursively write the embedded process instances.

⟨*Decays: decay term: TBP*⟩+≡
```
   procedure :: write_process_instances => decay_term_write_process_instances
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine decay_term_write_process_instances (term, unit, verbose)
     class(decay_term_t), intent(in) :: term
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose
     integer :: i
     do i = 1, size (term%particle_out)
        select type (unstable => term%particle_out(i)%c)
        type is (unstable_t)
           call unstable%write_process_instances (unit, verbose)
        end select
     end do
   end subroutine decay_term_write_process_instances
```

Initialization, using the configuration object. We allocate particle objects in
parallel to the particle configuration objects which we use to initialize them,
one at a time.

⟨*Decays: decay term: TBP*⟩+≡
```
   procedure :: init => decay_term_init
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine decay_term_init (term, config)
     class(decay_term_t), intent(out) :: term
     type(decay_term_config_t), intent(in), target :: config
     integer :: i
     term%config => config
     allocate (term%particle_out (size (config%prt)))
```

```
      do i = 1, size (config%prt)
         select type (prt_config => config%prt(i)%c)
         type is (stable_config_t)
            allocate (stable_t :: term%particle_out(i)%c)
            select type (stable => term%particle_out(i)%c)
            type is (stable_t)
               call stable%init (prt_config)
            end select
         type is (unstable_config_t)
            allocate (unstable_t :: term%particle_out(i)%c)
            select type (unstable => term%particle_out(i)%c)
            type is (unstable_t)
               call unstable%init (prt_config)
            end select
         end select
      end do
   end subroutine decay_term_init
```

Implement a RNG instance, spawned by the process object.

⟨*Decays: decay term: TBP*⟩+≡
```
      procedure :: make_rng => decay_term_make_rng
```

⟨*Decays: procedures*⟩+≡
```
   subroutine decay_term_make_rng (term, process)
     class(decay_term_t), intent(inout) :: term
     type(process_t), intent(inout) :: process
     class(rng_t), allocatable :: rng
     integer :: i
     do i = 1, size (term%particle_out)
        select type (unstable => term%particle_out(i)%c)
        type is (unstable_t)
           call process%make_rng (rng)
           call unstable%import_rng (rng)
        end select
     end do
   end subroutine decay_term_make_rng
```

Link the interactions for unstable decay products to the interaction of the parent process.

⟨*Decays: decay term: TBP*⟩+≡
```
      procedure :: link_interactions => decay_term_link_interactions
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine decay_term_link_interactions (term, trace)
     class(decay_term_t), intent(inout) :: term
     type(interaction_t), intent(in), target :: trace
     integer :: i
     do i = 1, size (term%particle_out)
        select type (unstable => term%particle_out(i)%c)
        type is (unstable_t)
           call unstable%link_interactions (i, trace)
        end select
     end do
   end subroutine decay_term_link_interactions
```

Recursively generate a decay chain, for each of the unstable particles in the final state.

⟨*Decays: decay term: TBP*⟩+≡
```
    procedure :: select_chain => decay_term_select_chain
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_term_select_chain (term)
      class(decay_term_t), intent(inout) :: term
      integer :: i
      do i = 1, size (term%particle_out)
         select type (unstable => term%particle_out(i)%c)
         type is (unstable_t)
            call unstable%select_chain ()
         end select
      end do
    end subroutine decay_term_select_chain
```

Recursively generate a decay event, for each of the unstable particles in the final state.

⟨*Decays: decay term: TBP*⟩+≡
```
    procedure :: generate => decay_term_generate
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_term_generate (term)
      class(decay_term_t), intent(inout) :: term
      integer :: i
      do i = 1, size (term%particle_out)
         select type (unstable => term%particle_out(i)%c)
         type is (unstable_t)
            call unstable%generate ()
         end select
      end do
    end subroutine decay_term_generate
```

17.2.5   Decay Root Configuration

At the root of a decay chain, there is a parent process. The decay root stores a pointer to the parent process and the set of decay configurations.

⟨*Decays: types*⟩+≡
```
    type :: decay_root_config_t
      type(string_t) :: process_id
      type(process_t), pointer :: process => null ()
      type(model_t), pointer :: model => null ()
      type(decay_term_config_t), dimension(:), allocatable :: term_config
     contains
     ⟨Decays: decay root config: TBP⟩
    end type decay_root_config_t
```

The finalizer is recursive since there may be cascade decays.

⟨*Decays: decay root config: TBP*⟩≡
```
    procedure :: final => decay_root_config_final
```

2335

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_root_config_final (object)
      class(decay_root_config_t), intent(inout) :: object
      integer :: i
      if (allocated (object%term_config)) then
         do i = 1, size (object%term_config)
            call object%term_config(i)%final ()
         end do
      end if
    end subroutine decay_root_config_final
```

The output routine is also recursive, and it contains an adjustable indentation.

⟨*Decays: decay root config: TBP*⟩+≡
```
    procedure :: write => decay_root_config_write
    procedure :: write_header => decay_root_config_write_header
    procedure :: write_terms => decay_root_config_write_terms
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_root_config_write (object, unit, indent, verbose)
      class(decay_root_config_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      logical, intent(in), optional :: verbose
      integer :: u, ind
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call write_indent (u, ind)
      write (u, "(1x,A)")  "Final-state decay tree:"
      call object%write_header (unit, indent)
      call object%write_terms (unit, indent, verbose)
    end subroutine decay_root_config_write

    subroutine decay_root_config_write_header (object, unit, indent)
      class(decay_root_config_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      integer :: u, ind
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call write_indent (u, ind)
      if (associated (object%process)) then
         write (u, 3)  "process ID      =", char (object%process_id), "*"
      else
         write (u, 3)  "process ID      =", char (object%process_id)
      end if
  3   format (3x,A,2(1x,A))
    end subroutine decay_root_config_write_header

    recursive subroutine decay_root_config_write_terms &
         (object, unit, indent, verbose)
      class(decay_root_config_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      logical, intent(in), optional :: verbose
      integer :: i, u, ind
      logical :: verb
      u = output_unit (unit)
```

```
        ind = 0;  if (present (indent))  ind = indent
        verb = .true.;  if (present (verbose))  verb = verbose
        if (verb .and. allocated (object%term_config)) then
           do i = 1, size (object%term_config)
              call object%term_config(i)%write (u, ind + 1)
           end do
        end if
     end subroutine decay_root_config_write_terms
```

Initialize for a named process and (optionally) a pre-determined number of terms.

⟨*Decays: decay root config: TBP*⟩+≡
```
     procedure :: init => decay_root_config_init
```

⟨*Decays: procedures*⟩+≡
```
     subroutine decay_root_config_init (decay, model, process_id, n_terms)
       class(decay_root_config_t), intent(out) :: decay
       type(model_t), intent(in), target :: model
       type(string_t), intent(in) :: process_id
       integer, intent(in), optional :: n_terms
       decay%model => model
       decay%process_id = process_id
       if (present (n_terms)) then
          allocate (decay%term_config (n_terms))
       end if
     end subroutine decay_root_config_init
```

Declare a decay term, given an array of flavors.

⟨*Decays: decay root config: TBP*⟩+≡
```
     procedure :: init_term => decay_root_config_init_term
```

⟨*Decays: procedures*⟩+≡
```
     recursive subroutine decay_root_config_init_term &
          (decay, i, flv, stable, model, process_stack)
       class(decay_root_config_t), intent(inout) :: decay
       integer, intent(in) :: i
       type(flavor_t), dimension(:,:), intent(in) :: flv
       logical, dimension(:), intent(in) :: stable
       type(model_t), intent(in), target :: model
       type(process_stack_t), intent(in), optional :: process_stack
       call decay%term_config(i)%init (flv, stable, model, process_stack)
     end subroutine decay_root_config_init_term
```

Connect the decay root configuration with a process object (which should represent the parent process). This includes initialization, therefore intent(out).

The flavor state is retrieved from the process term object. However, we have to be careful: the flavor object points to the model instance that is stored in the process object. This model instance may not contain the current setting for unstable particles and decay. Therefore, we assign the model directly.

If the `process_instance` argument is provided, we use this for the flavor state. This applies to the decay root only, where the process can be entangled with a beam setup, and the latter contains beam remnants as further outgoing

particles. These must be included in the set of outgoing flavors, since the decay application is also done on the connected state.

Infer stability from the particle properties, using the first row in the set of flavor states. For unstable particles, we look for decays, recursively, available from the process stack (if present).

⟨*Decays: decay root config: TBP*⟩+≡
```
    procedure :: connect => decay_root_config_connect
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_root_config_connect &
         (decay, process, model, process_stack, process_instance)
      class(decay_root_config_t), intent(out) :: decay
      type(process_t), intent(in), target :: process
      type(model_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      type(process_instance_t), intent(in), optional, target :: process_instance
      type(connected_state_t), pointer :: connected_state
      type(interaction_t), pointer :: int
      type(flavor_t), dimension(:,:), allocatable :: flv
      logical, dimension(:), allocatable :: stable
      integer :: i
      call decay%init (model, process%get_id (), process%get_n_terms ())
      do i = 1, size (decay%term_config)
         if (present (process_instance)) then
            connected_state => process_instance%get_connected_state_ptr (i)
            int => connected_state%get_matrix_int_ptr ()
            call interaction_get_flv_out (int, flv)
         else
            call process%get_term_flv_out (i, flv)
         end if
         call flavor_set_model (flv, model)
         allocate (stable (size (flv, 1)))
         stable = flavor_is_stable (flv(:,1))
         call decay%init_term (i, flv, stable, model, process_stack)
         deallocate (flv, stable)
      end do
      decay%process => process
    end subroutine decay_root_config_connect
```

Recursively compute widths, errors, and branching ratios.

⟨*Decays: decay root config: TBP*⟩+≡
```
    procedure :: compute => decay_root_config_compute
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_root_config_compute (decay)
      class(decay_root_config_t), intent(inout) :: decay
      integer :: i
      do i = 1, size (decay%term_config)
         call decay%term_config(i)%compute ()
      end do
    end subroutine decay_root_config_compute
```

17.2.6 Decay Root Instance

This is the common parent type for decay and decay root. The process instance points to the parent process. The model pointer is separate because particle settings may be updated w.r.t. the parent process object.

⟨*Decays: types*⟩+≡
```
type, abstract :: decay_gen_t
  type(decay_term_t), dimension(:), allocatable :: term
  type(process_instance_t), pointer :: process_instance => null ()
  integer :: selected_mci = 0
  integer :: selected_term = 0
 contains
 ⟨Decays: decay gen: TBP⟩
end type decay_gen_t
```

The decay root represents the parent process. When an event is generated, the generator selects the term to which the decay chain applies (if possible).

The process instance is just a pointer.

⟨*Decays: types*⟩+≡
```
type, extends (decay_gen_t) :: decay_root_t
  type(decay_root_config_t), pointer :: config => null ()
 contains
 ⟨Decays: decay root: TBP⟩
end type decay_root_t
```

The finalizer has to recursively finalize the terms, but we can skip the process instance which is not explicitly allocated.

⟨*Decays: decay gen: TBP*⟩≡
```
procedure :: base_final => decay_gen_final
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine decay_gen_final (object)
  class(decay_gen_t), intent(inout) :: object
  integer :: i
  if (allocated (object%term)) then
     do i = 1, size (object%term)
        call object%term(i)%final ()
     end do
  end if
end subroutine decay_gen_final
```

No extra finalization for the decay root.

⟨*Decays: decay root: TBP*⟩≡
```
procedure :: final => decay_root_final
```

⟨*Decays: procedures*⟩+≡
```
subroutine decay_root_final (object)
  class(decay_root_t), intent(inout) :: object
  call object%base_final ()
end subroutine decay_root_final
```

Output.

⟨*Decays: decay root: TBP*⟩+≡
```
    procedure :: write => decay_root_write
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_root_write (object, unit)
      class(decay_root_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (associated (object%config)) then
         call object%config%write (unit, verbose = .false.)
      else
         write (u, "(1x,A)")  "Final-state decay tree: [not configured]"
      end if
      if (object%selected_mci > 0) then
         write (u, "(3x,A,I0)")  "Selected MCI    = ", object%selected_mci
      else
         write (u, "(3x,A)")  "Selected MCI    = [undefined]"
      end if
      if (object%selected_term > 0) then
         write (u, "(3x,A,I0)")  "Selected term   = ", object%selected_term
         call object%term(object%selected_term)%write (u, 1)
      else
         write (u, "(3x,A)")  "Selected term   = [undefined]"
      end if
    end subroutine decay_root_write
```

Write the process instances, recursively.

⟨*Decays: decay gen: TBP*⟩+≡
```
    procedure :: write_process_instances => decay_gen_write_process_instances
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_gen_write_process_instances (decay, unit, verbose)
      class(decay_gen_t), intent(in) :: decay
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      logical :: verb
      verb = .true.;  if (present (verbose))  verb = verbose
      if (associated (decay%process_instance)) then
         if (verb) then
            call decay%process_instance%write (unit)
         else
            call decay%process_instance%write_header (unit)
         end if
      end if
      if (decay%selected_term > 0) then
         call decay%term(decay%selected_term)%write_process_instances (unit, verb)
      end if
    end subroutine decay_gen_write_process_instances
```

Generic initializer. All can be done recursively.

⟨*Decays: decay gen: TBP*⟩+≡
```
    procedure :: base_init => decay_gen_init
```

⟨*Decays: procedures*⟩+≡

```
    recursive subroutine decay_gen_init (decay, term_config)
      class(decay_gen_t), intent(out) :: decay
      type(decay_term_config_t), dimension(:), intent(in), target :: term_config
      integer :: i
      allocate (decay%term (size (term_config)))
      do i = 1, size (decay%term)
         call decay%term(i)%init (term_config(i))
      end do
    end subroutine decay_gen_init
```

Specific initializer. We assign the configuration object, which should correspond
to a completely initialized decay configuration tree. We also connect to an
existing process instance. Then, we recursively link the child interactions to the
parent process.

⟨*Decays: decay root: TBP*⟩+≡

```
    procedure :: init => decay_root_init
```

⟨*Decays: procedures*⟩+≡

```
    subroutine decay_root_init (decay_root, config, process_instance)
      class(decay_root_t), intent(out) :: decay_root
      type(decay_root_config_t), intent(in), target :: config
      type(process_instance_t), intent(in), target :: process_instance
      call decay_root%base_init (config%term_config)
      decay_root%config => config
      decay_root%process_instance => process_instance
      call decay_root%make_term_rng (config%process)
      call decay_root%link_term_interactions ()
    end subroutine decay_root_init
```

Implement random-number generators for unstable decay selection in all terms.
This is not recursive.

   We also make use of the fact that `process` is a pointer; the (state of the
RNG factory inside the) target process will be modified by the rng-spawning
method, but not the pointer.

⟨*Decays: decay gen: TBP*⟩+≡

```
    procedure :: make_term_rng => decay_gen_make_term_rng
```

⟨*Decays: procedures*⟩+≡

```
    subroutine decay_gen_make_term_rng (decay, process)
      class(decay_gen_t), intent(inout) :: decay
      type(process_t), intent(in), pointer :: process
      integer :: i
      do i = 1, size (decay%term)
         call decay%term(i)%make_rng (process)
      end do
    end subroutine decay_gen_make_term_rng
```

Recursively link interactions of the enclosed decay terms to the corresponding
terms in the current process instance.

   Note: A bug in nagfor requires the extra `i_term` variable.

⟨*Decays: decay gen: TBP*⟩+≡

```
    procedure :: link_term_interactions => decay_gen_link_term_interactions
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_gen_link_term_interactions (decay)
      class(decay_gen_t), intent(inout) :: decay
      integer :: i, i_term
      type(interaction_t), pointer :: trace
      associate (instance => decay%process_instance)
        do i = 1, size (decay%term)
           i_term = i
           trace => instance%get_trace_int_ptr (i_term)
           call decay%term(i_term)%link_interactions (trace)
        end do
      end associate
    end subroutine decay_gen_link_term_interactions
```

Select a decay chain: decay modes and process components.

⟨*Decays: decay root: TBP*⟩+≡
```
    procedure :: select_chain => decay_root_select_chain
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_root_select_chain (decay_root)
      class(decay_root_t), intent(inout) :: decay_root
      if (decay_root%selected_term > 0) then
         call decay_root%term(decay_root%selected_term)%select_chain ()
      else
         call msg_bug ("Decays: no term selected for parent process")
      end if
    end subroutine decay_root_select_chain
```

Generate a decay tree, i.e., for the selected term in the parent process, recursively generate a decay event for all unstable particles.

Factor out the trace of the connected state of the parent process. This trace should not be taken into account for unweighting the decay chain, since it was already used for unweighting the parent event, or it determines the overall event weight.

⟨*Decays: decay root: TBP*⟩+≡
```
    procedure :: generate => decay_root_generate
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_root_generate (decay_root)
      class(decay_root_t), intent(inout) :: decay_root
      type(connected_state_t), pointer :: connected_state
      if (decay_root%selected_term > 0) then
         connected_state => decay_root%process_instance%get_connected_state_ptr &
              (decay_root%selected_term)
         call connected_state%normalize_matrix_by_trace ()
         call decay_root%term(decay_root%selected_term)%generate ()
      else
         call msg_bug ("Decays: no term selected for parent process")
      end if
    end subroutine decay_root_generate
```

17.2.7 Decay Configuration

A decay configuration describes a distinct decay mode of a particle. Each decay mode may include several terms, which correspond to the terms in the associated process. In addition to the base type, the decay configuration object contains the integral of the parent process and the selector for the MCI group inside this process.

⟨*Decays: types*⟩+≡
```
type, extends (decay_root_config_t) :: decay_config_t
  real(default) :: weight = 0
  real(default) :: integral = 0
  real(default) :: abs_error = 0
  real(default) :: rel_error = 0
  type(selector_t) :: mci_selector
 contains
 ⟨Decays: decay config: TBP⟩
 end type decay_config_t
```

The output routine extends the decay-root writer by listing numerical component values.

⟨*Decays: decay config: TBP*⟩≡
```
procedure :: write => decay_config_write
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine decay_config_write (object, unit, indent, verbose)
  class(decay_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  logical, intent(in), optional :: verbose
  integer :: u, ind
  u = output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  write (u, "(1x,A)")  "Decay:"
  call object%write_header (unit, indent)
  call write_indent (u, ind)
  write (u, 2)  "branching ratio =", object%weight * 100
  call write_indent (u, ind)
  write (u, 1)  "partial width   =", object%integral
  call write_indent (u, ind)
  write (u, 1)  "error (abs)     =", object%abs_error
  call write_indent (u, ind)
  write (u, 1)  "error (rel)     =", object%rel_error
1   format (3x,A,ES19.12)
2   format (3x,A,F11.6,1x,'%')
  call object%write_terms (unit, indent, verbose)
 end subroutine decay_config_write
```

Connect a decay configuration with a process object (which should represent the decay). This includes initialization, therefore intent(out). We first connect the process itself, then do initializations that are specific for this decay.

Infer stability from the particle properties, using the first row in the set of flavor states. Once we can deal with predetermined decay chains, they should

be used instead.

⟨*Decays: decay config: TBP*⟩+≡
```
    procedure :: connect => decay_config_connect
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_config_connect &
         (decay, process, model, process_stack, process_instance)
      class(decay_config_t), intent(out) :: decay
      type(process_t), intent(in), target :: process
      type(model_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      type(process_instance_t), intent(in), optional, target :: process_instance
      real(default), dimension(:), allocatable :: integral_mci
      integer :: i, n_mci
      call decay%decay_root_config_t%connect (process, model, process_stack)
      decay%integral = process%get_integral ()
      decay%abs_error = process%get_error ()
      if (process%cm_frame ()) then
         call msg_fatal ("Decay process " // char (process%get_id ()) &
              // ": unusable because rest frame is fixed.")
      end if
      n_mci = process%get_n_mci ()
      allocate (integral_mci (n_mci))
      do i = 1, n_mci
         integral_mci(i) = process%get_integral_mci (i)
      end do
      call decay%mci_selector%init (integral_mci)
    end subroutine decay_config_connect
```

Compute embedded branchings and the relative error. This method does not
apply to the decay root.

⟨*Decays: decay config: TBP*⟩+≡
```
    procedure :: compute => decay_config_compute
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_config_compute (decay)
      class(decay_config_t), intent(inout) :: decay
      call decay%decay_root_config_t%compute ()
      if (decay%integral == 0) then
         call decay%write ()
         call msg_fatal ("Decay configuration: partial width is zero")
      end if
      decay%rel_error = decay%abs_error / decay%integral
    end subroutine decay_config_compute
```

17.2.8 Decay Instance

The decay contains a collection of terms. One of them is selected when the
decay is evaluated. This is similar to the decay root, but we implement it
independently.

The process instance object is allocated via a pointer, so it automatically
behaves as a target.

⟨*Decays: types*⟩+≡

```
    type, extends (decay_gen_t) :: decay_t
      type(decay_config_t), pointer :: config => null ()
      class(rng_t), allocatable :: rng
     contains
     ⟨Decays: decay: TBP⟩
    end type decay_t
```

The finalizer is recursive.

⟨*Decays: decay: TBP*⟩≡
```
    procedure :: final => decay_final
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_final (object)
      class(decay_t), intent(inout) :: object
      integer :: i
      call object%base_final ()
      do i = 1, object%config%process%get_n_mci ()
         call object%process_instance%final_simulation (i)
      end do
      call object%process_instance%final ()
      deallocate (object%process_instance)
    end subroutine decay_final
```

Output.

⟨*Decays: decay: TBP*⟩+≡
```
    procedure :: write => decay_write
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_write (object, unit, indent, recursive)
      class(decay_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent, recursive
      integer :: u, ind
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call object%config%write (unit, indent, verbose = .false.)
      if (allocated (object%rng)) then
         call object%rng%write (u, ind + 1)
      end if
      call write_indent (u, ind)
      if (object%selected_mci > 0) then
         write (u, "(3x,A,I0)")  "Selected MCI   = ", object%selected_mci
      else
         write (u, "(3x,A)")  "Selected MCI   = [undefined]"
      end if
      call write_indent (u, ind)
      if (object%selected_term > 0) then
         write (u, "(3x,A,I0)")  "Selected term  = ", object%selected_term
         call object%term(object%selected_term)%write (u, ind + 1)
      else
         write (u, "(3x,A)")  "Selected term  = [undefined]"
      end if
    end subroutine decay_write
```

Initializer. Base initialization is done recursively. Then, we prepare the current process instance and allocate a random-number generator for term selection. For all unstable particles, we also allocate a r.n.g. as spawned by the current process.

⟨*Decays: decay: TBP*⟩+≡
```
    procedure :: init => decay_init
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_init (decay, config)
      class(decay_t), intent(out) :: decay
      type(decay_config_t), intent(in), target :: config
      integer :: i
      call decay%base_init (config%term_config)
      decay%config => config
      allocate (decay%process_instance)
      call decay%process_instance%init (decay%config%process)
      call decay%process_instance%setup_event_data (decay%config%model)
      do i = 1, decay%config%process%get_n_mci ()
         call decay%process_instance%init_simulation (i)
      end do
      call decay%config%process%make_rng (decay%rng)
      call decay%make_term_rng (decay%config%process)
    end subroutine decay_init
```

Link interactions to the parent process. `i_prt` is the index of the current outgoing particle in the parent interaction, for which we take the trace evaluator. We link it to the beam particle in the beam interaction of the decay process instance. Then, repeat the procedure for the outgoing particles.

⟨*Decays: decay: TBP*⟩+≡
```
    procedure :: link_interactions => decay_link_interactions
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_link_interactions (decay, i_prt, trace)
      class(decay_t), intent(inout) :: decay
      integer, intent(in) :: i_prt
      type(interaction_t), intent(in), target :: trace
      type(interaction_t), pointer :: beam_int
      integer :: n_in, n_vir
      beam_int => decay%process_instance%get_beam_int_ptr ()
      n_in = interaction_get_n_in (trace)
      n_vir = interaction_get_n_vir (trace)
      call interaction_set_source_link (beam_int, 1, trace, &
           n_in + n_vir + i_prt)
      call decay%link_term_interactions ()
    end subroutine decay_link_interactions
```

Determine a decay chain. For each unstable particle we select one of the possible decay modes, and for each decay process we select one of the possible decay MCI components, calling the random-number generators. We do not generate momenta, yet.

⟨*Decays: decay: TBP*⟩+≡
```
    procedure :: select_chain => decay_select_chain
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine decay_select_chain (decay)
  class(decay_t), intent(inout) :: decay
  real(default) :: x
  integer :: i
  call decay%rng%generate (x)
  decay%selected_mci = decay%config%mci_selector%select (x)
  call decay%process_instance%choose_mci (decay%selected_mci)
  call decay%process_instance%select_i_term (decay%selected_term)
  do i = 1, size (decay%term)
     call decay%term(i)%select_chain ()
  end do
end subroutine decay_select_chain
```

Generate a decay. We first receive the beam momenta from the parent process (assuming that this is properly linked), then call the associated process object for a new event.

Factor out the trace of the helicity density matrix of the isolated state (the one that will be used for the decay chain). The trace is taken into account for unweighting the individual decay event and should therefore be ignored for unweighting the correlated decay chain afterwards.

⟨*Decays: decay: TBP*⟩+≡
```
procedure :: generate => decay_generate
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine decay_generate (decay)
  class(decay_t), intent(inout) :: decay
  type(isolated_state_t), pointer :: isolated_state
  integer :: i
  call decay%process_instance%receive_beam_momenta ()
  call decay%config%process%generate_unweighted_event &
       (decay%process_instance, decay%selected_mci)
  if (signal_is_pending ())  return
  call decay%process_instance%evaluate_event_data ()
  isolated_state => &
       decay%process_instance%get_isolated_state_ptr (decay%selected_term)
  call isolated_state%normalize_matrix_by_trace ()
  do i = 1, size (decay%term)
     call decay%term(i)%generate ()
     if (signal_is_pending ())  return
  end do
end subroutine decay_generate
```

17.2.9   Stable Particles

This is a stable particle. The flavor can be ambiguous (e.g., partons).

⟨*Decays: types*⟩+≡
```
type, extends (any_config_t) :: stable_config_t
  type(flavor_t), dimension(:), allocatable :: flv
 contains
  ⟨Decays: stable config: TBP⟩
```

```
      end type stable_config_t
```

The finalizer is empty:

⟨Decays: stable config: TBP⟩≡
```
    procedure :: final => stable_config_final
```

⟨Decays: procedures⟩+≡
```
    subroutine stable_config_final (object)
      class(stable_config_t), intent(inout) :: object
    end subroutine stable_config_final
```

Output.

⟨Decays: stable config: TBP⟩+≡
```
    procedure :: write => stable_config_write
```

⟨Decays: procedures⟩+≡
```
    recursive subroutine stable_config_write (object, unit, indent, verbose)
      class(stable_config_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      logical, intent(in), optional :: verbose
      integer :: u, i, ind
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call write_indent (u, ind)
      write (u, "(1x,'+',1x,A)", advance = "no")  "Stable:"
      write (u, "(1x,A)", advance = "no")  char (flavor_get_name (object%flv(1)))
      do i = 2, size (object%flv)
         write (u, "(':',A)", advance = "no") &
              char (flavor_get_name (object%flv(i)))
      end do
      write (u, *)
    end subroutine stable_config_write
```

Initializer. We are presented with an array of flavors, but there may be double
entries which we remove, so we store only the distinct flavors.

⟨Decays: stable config: TBP⟩+≡
```
    procedure :: init => stable_config_init
```

⟨Decays: procedures⟩+≡
```
    subroutine stable_config_init (config, flv)
      class(stable_config_t), intent(out) :: config
      type(flavor_t), dimension(:), intent(in) :: flv
      integer, dimension (size (flv)) :: pdg
      logical, dimension (size (flv)) :: mask
      integer :: i
      pdg = flavor_get_pdg (flv)
      mask(1) = .true.
      forall (i = 2 : size (pdg))
         mask(i) = all (pdg(i) /= pdg(1:i-1))
      end forall
      allocate (config%flv (count (mask)))
      config%flv = pack (flv, mask)
    end subroutine stable_config_init
```

Here is the corresponding object instance. Except for the pointer to the configuration, there is no content.

⟨*Decays: types*⟩+≡
```
type, extends (any_t) :: stable_t
  type(stable_config_t), pointer :: config => null ()
 contains
 ⟨Decays: stable: TBP⟩
end type stable_t
```

The finalizer does nothing.

⟨*Decays: stable: TBP*⟩≡
```
procedure :: final => stable_final
```

⟨*Decays: procedures*⟩+≡
```
subroutine stable_final (object)
  class(stable_t), intent(inout) :: object
end subroutine stable_final
```

We can delegate output to the configuration object.

⟨*Decays: stable: TBP*⟩+≡
```
procedure :: write => stable_write
```

⟨*Decays: procedures*⟩+≡
```
subroutine stable_write (object, unit, indent)
  class(stable_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  call object%config%write (unit, indent)
end subroutine stable_write
```

Initializer: just assign the configuration.

⟨*Decays: stable: TBP*⟩+≡
```
procedure :: init => stable_init
```

⟨*Decays: procedures*⟩+≡
```
subroutine stable_init (stable, config)
  class(stable_t), intent(out) :: stable
  type(stable_config_t), intent(in), target :: config
  stable%config => config
end subroutine stable_init
```

17.2.10 Unstable Particles

A branching configuration enables us to select among distinct decay modes of a particle. We store the particle flavor (with its implicit link to a model), an array of decay configurations, and a selector object.

The total width, absolute and relative error are stored as `integral`, `abs_error`, and `rel_error`, respectively.

The flavor must be unique in this case.

⟨*Decays: types*⟩+≡
```
type, extends (any_config_t) :: unstable_config_t
  type(flavor_t) :: flv
```

```
      real(default) :: integral = 0
      real(default) :: abs_error = 0
      real(default) :: rel_error = 0
      type(selector_t) :: selector
      type(decay_config_t), dimension(:), allocatable :: decay_config
    contains
    ⟨Decays: unstable config: TBP⟩
    end type unstable_config_t
```

Finalizer. The branching configuration can be a recursive structure.

⟨*Decays: unstable config: TBP*⟩≡
```
    procedure :: final => unstable_config_final
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_config_final (object)
      class(unstable_config_t), intent(inout) :: object
      integer :: i
      if (allocated (object%decay_config)) then
         do i = 1, size (object%decay_config)
            call object%decay_config(i)%final ()
         end do
      end if
    end subroutine unstable_config_final
```

Output. Since this may be recursive, we include indentation.

⟨*Decays: unstable config: TBP*⟩+≡
```
    procedure :: write => unstable_config_write
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_config_write (object, unit, indent, verbose)
      class(unstable_config_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      logical, intent(in), optional :: verbose
      integer :: u, i, ind
      logical :: verb
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      verb = .true.;  if (present (verbose))  verb = verbose
      call write_indent (u, ind)
      write (u, "(1x,'+',1x,A,1x,A)")  "Unstable:", &
           char (flavor_get_name (object%flv))
      call write_indent (u, ind)
      write (u, 1)  "total width =", object%integral
      call write_indent (u, ind)
      write (u, 1)  "error (abs) =", object%abs_error
      call write_indent (u, ind)
      write (u, 1)  "error (rel) =", object%rel_error
  1   format (5x,A,ES19.12)
      if (verb .and. allocated (object%decay_config)) then
         do i = 1, size (object%decay_config)
            call object%decay_config(i)%write (u, ind + 1)
         end do
      end if
    end subroutine unstable_config_write
```

2350

Initializer. For the unstable particle, the flavor is unique.

⟨*Decays: unstable config: TBP*⟩+≡
```
    procedure :: init => unstable_config_init
```

⟨*Decays: procedures*⟩+≡
```
    subroutine unstable_config_init (config, flv)
      class(unstable_config_t), intent(out) :: config
      type(flavor_t), intent(in) :: flv
      config%flv = flv
    end subroutine unstable_config_init
```

Further initialization: determine the number of decay modes. We can assume that the flavor of the particle has been set already.

If the process stack is given, we can delve recursively into actually assigning decay processes. Otherwise, we just initialize with decay process names.

⟨*Decays: unstable config: TBP*⟩+≡
```
    procedure :: init_decays => unstable_config_init_decays
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_config_init_decays &
        (unstable, decay_id, model, process_stack)
      class(unstable_config_t), intent(inout) :: unstable
      type(string_t), dimension(:), intent(in) :: decay_id
      type(model_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      integer :: i
      allocate (unstable%decay_config (size (decay_id)))
      do i = 1, size (decay_id)
         associate (decay => unstable%decay_config(i))
           if (present (process_stack)) then
              call decay%connect (process_stack%get_process_ptr (decay_id(i)), &
                   model, process_stack)
           else
              call decay%init (model, decay_id(i))
           end if
         end associate
      end do
    end subroutine unstable_config_init_decays
```

Compute the total width and branching ratios, initializing the decay selector.

⟨*Decays: unstable config: TBP*⟩+≡
```
    procedure :: compute => unstable_config_compute
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_config_compute (unstable)
      class(unstable_config_t), intent(inout) :: unstable
      integer :: i
      do i = 1, size (unstable%decay_config)
         call unstable%decay_config(i)%compute ()
      end do
      unstable%integral = sum (unstable%decay_config%integral)
      if (unstable%integral <= 0) then
```

```
            call unstable%write ()
            call msg_fatal ("Decay configuration: computed total width is zero")
         end if
         unstable%abs_error = sqrt (sum (unstable%decay_config%abs_error ** 2))
         unstable%rel_error = unstable%abs_error / unstable%integral
         call unstable%selector%init (unstable%decay_config%integral)
         do i = 1, size (unstable%decay_config)
            unstable%decay_config(i)%weight &
                 = unstable%selector%get_weight (i)
         end do
    end subroutine unstable_config_compute
```

Now we define the instance of an unstable particle.

⟨*Decays: types*⟩+≡
```
    type, extends (any_t) :: unstable_t
       type(unstable_config_t), pointer :: config => null ()
       class(rng_t), allocatable :: rng
       integer :: selected_decay = 0
       type(decay_t), dimension(:), allocatable :: decay
     contains
     ⟨Decays: unstable: TBP⟩
    end type unstable_t
```

Recursive finalizer.

⟨*Decays: unstable: TBP*⟩≡
```
    procedure :: final => unstable_final
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_final (object)
      class(unstable_t), intent(inout) :: object
      integer :: i
      if (allocated (object%decay)) then
         do i = 1, size (object%decay)
            call object%decay(i)%final ()
         end do
      end if
    end subroutine unstable_final
```

Output.

⟨*Decays: unstable: TBP*⟩+≡
```
    procedure :: write => unstable_write
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_write (object, unit, indent)
      class(unstable_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
      integer :: u, ind
      u = output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call object%config%write (u, ind, verbose=.false.)
      if (allocated (object%rng)) then
         call object%rng%write (u, ind + 2)
      end if
```

```
      call write_indent (u, ind)
      if (object%selected_decay > 0) then
         write (u, "(5x,A,I0)") "Sel. decay  = ", object%selected_decay
         call object%decay(object%selected_decay)%write (u, ind + 1)
      else
         write (u, "(5x,A)")  "Sel. decay  = [undefined]"
      end if
   end subroutine unstable_write
```

Write the embedded process instances.

⟨*Decays: unstable: TBP*⟩+≡
```
   procedure :: write_process_instances => unstable_write_process_instances
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine unstable_write_process_instances &
        (unstable, unit, verbose)
     class(unstable_t), intent(in) :: unstable
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose
     if (unstable%selected_decay > 0) then
        call unstable%decay(unstable%selected_decay)% &
             write_process_instances (unit, verbose)
     end if
   end subroutine unstable_write_process_instances
```

Initialization, using the configuration object.

⟨*Decays: unstable: TBP*⟩+≡
```
   procedure :: init => unstable_init
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine unstable_init (unstable, config)
     class(unstable_t), intent(out) :: unstable
     type(unstable_config_t), intent(in), target :: config
     integer :: i
     unstable%config => config
     allocate (unstable%decay (size (config%decay_config)))
     do i = 1, size (config%decay_config)
        call unstable%decay(i)%init (config%decay_config(i))
     end do
   end subroutine unstable_init
```

Recursively link interactions to the parent process. `i_prt` is the index of the
current outgoing particle in the parent interaction.

⟨*Decays: unstable: TBP*⟩+≡
```
   procedure :: link_interactions => unstable_link_interactions
```

⟨*Decays: procedures*⟩+≡
```
   recursive subroutine unstable_link_interactions (unstable, i_prt, trace)
     class(unstable_t), intent(inout) :: unstable
     integer, intent(in) :: i_prt
     type(interaction_t), intent(in), target :: trace
     integer :: i
     do i = 1, size (unstable%decay)
```

```
        call unstable%decay(i)%link_interactions (i_prt, trace)
      end do
    end subroutine unstable_link_interactions
```

Import the random-number generator state.

⟨*Decays: unstable: TBP*⟩+≡
```
    procedure :: import_rng => unstable_import_rng
```

⟨*Decays: procedures*⟩+≡
```
    subroutine unstable_import_rng (unstable, rng)
      class(unstable_t), intent(inout) :: unstable
      class(rng_t), intent(inout), allocatable :: rng
      call move_alloc (from = rng, to = unstable%rng)
    end subroutine unstable_import_rng
```

Generate a decay chain. First select a decay mode, then call the `select_chain` method of the selected mode.

⟨*Decays: unstable: TBP*⟩+≡
```
    procedure :: select_chain => unstable_select_chain
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_select_chain (unstable)
      class(unstable_t), intent(inout) :: unstable
      real(default) :: x
      call unstable%rng%generate (x)
      unstable%selected_decay = unstable%config%selector%select (x)
      call unstable%decay(unstable%selected_decay)%select_chain ()
    end subroutine unstable_select_chain
```

Generate a decay event.

⟨*Decays: unstable: TBP*⟩+≡
```
    procedure :: generate => unstable_generate
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine unstable_generate (unstable)
      class(unstable_t), intent(inout) :: unstable
      call unstable%decay(unstable%selected_decay)%generate ()
    end subroutine unstable_generate
```

17.2.11   Decay Chain

While the decay configuration tree and the decay tree are static entities (during a simulation run), the decay chain is dynamically generated for each event. The reason is that with the possibility of several decay modes for each particle, and several terms for each process, the total number of distinct decay chains is not under control.

Each entry in the decay chain is a connected parton state. The origin of the chain is a connected state in the parent process (not part of the chain itself). For each decay, mode and term chosen, we convolute this with the isolated (!) state of the current decay, to generate a new connected state. We accumulate this chain by recursively traversing the allocated decay tree. Whenever a particle

decays, it becomes virtual and is replaced by its decay product, while all other particles stay in the parton state as spectators.

Technically, we implement the decay chain as a stack structure and include information from the associated decay object for easier debugging. This is a decay chain entry:

⟨*Decays: types*⟩+≡
```
type, extends (connected_state_t) :: decay_chain_entry_t
  integer :: index = 0
  type(decay_config_t), pointer :: config => null ()
  integer :: selected_mci = 0
  integer :: selected_term = 0
  type(decay_chain_entry_t), pointer :: previous => null ()
end type decay_chain_entry_t
```

This is the complete chain; we need just a pointer to the last entry. We also include a pointer to the master process instance, which serves as the seed for the decay chain.

The evaluator `correlated_trace` traces over all quantum numbers for the final spin-correlated (but color-summed) evaluator of the decay chain. This allows us to compute the probability for a momentum configuration, given that all individual density matrices (of the initial process and the subsequent decays) have been normalized to one.

Note: This trace is summed over color, so color is treated exactly when computing spin correlations. However, we do not keep non-diagonal color correlations. When an event is accepted, we compute probabilities for all color states and can choose one of them.

⟨*Decays: types*⟩+≡
```
type :: decay_chain_t
  type(process_instance_t), pointer :: process_instance => null ()
  integer :: selected_term = 0
  type(evaluator_t) :: correlated_trace
  type(decay_chain_entry_t), pointer :: last => null ()
contains
⟨Decays: decay chain: TBP⟩
end type decay_chain_t
```

The finalizer recursively deletes and deallocates the entries.

⟨*Decays: decay chain: TBP*⟩≡
```
procedure :: final => decay_chain_final
```

⟨*Decays: procedures*⟩+≡
```
subroutine decay_chain_final (object)
  class(decay_chain_t), intent(inout) :: object
  type(decay_chain_entry_t), pointer :: entry
  do while (associated (object%last))
    entry => object%last
    object%last => entry%previous
    call entry%final ()
    deallocate (entry)
  end do
  call evaluator_final (object%correlated_trace)
end subroutine decay_chain_final
```

Doing output recursively allows us to display the chain in chronological order.

⟨*Decays: decay chain: TBP*⟩+≡
```
    procedure :: write => decay_chain_write
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_chain_write (object, unit)
      class(decay_chain_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      call write_separator_double (u)
      write (u, "(1x,A)")  "Decay chain:"
      call write_entries (object%last)
      call write_separator_double (u)
      write (u, "(1x,A)")  "Evaluator (correlated trace of the decay chain):"
      call write_separator (u)
      call object%correlated_trace%write (u)
      call write_separator_double (u)
    contains
      recursive subroutine write_entries (entry)
        type(decay_chain_entry_t), intent(in), pointer :: entry
        if (associated (entry)) then
           call write_entries (entry%previous)
           call write_separator_double (u)
           write (u, "(1x,A,I0)")  "Decay #", entry%index
           call entry%config%write_header (u)
           write (u, "(3x,A,I0)")  "Selected MCI   = ", entry%selected_mci
           write (u, "(3x,A,I0)")  "Selected term  = ", entry%selected_term
           call entry%config%term_config(entry%selected_term)%write (u, indent=1)
           call entry%write (u)
        end if
      end subroutine write_entries
    end subroutine decay_chain_write
```

Build a decay chain, recursively following the selected decays and terms in a decay tree. Before start, we finalize the chain, deleting any previous contents.

⟨*Decays: decay chain: TBP*⟩+≡
```
    procedure :: build => decay_chain_build
```

⟨*Decays: procedures*⟩+≡
```
    subroutine decay_chain_build (chain, decay_root)
      class(decay_chain_t), intent(inout), target :: chain
      type(decay_root_t), intent(in) :: decay_root
      type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
      type(interaction_t), pointer :: int_last_decay
      call chain%final ()
      if (decay_root%selected_term > 0) then
         chain%process_instance => decay_root%process_instance
         chain%selected_term = decay_root%selected_term
         call chain%build_term_entries (decay_root%term(decay_root%selected_term))
      end if
      int_last_decay => chain%last%get_matrix_int_ptr ()
      allocate (qn_mask (interaction_get_n_tot (int_last_decay)))
```

```
      call quantum_numbers_mask_init (qn_mask, &
          mask_f = .true., mask_c = .true., mask_h = .true.)
      call evaluator_init_qn_sum (chain%correlated_trace, int_last_decay, qn_mask)
    end subroutine decay_chain_build
```

Build the entries that correspond to a decay term. We have to scan all unstable
particles.

⟨*Decays: decay chain: TBP*⟩+≡
```
    procedure :: build_term_entries => decay_chain_build_term_entries
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_chain_build_term_entries (chain, term)
      class(decay_chain_t), intent(inout) :: chain
      type(decay_term_t), intent(in) :: term
      integer :: i
      do i = 1, size (term%particle_out)
         select type (unstable => term%particle_out(i)%c)
         type is (unstable_t)
            if (unstable%selected_decay > 0) then
               call chain%build_decay_entries &
                    (unstable%decay(unstable%selected_decay))
            end if
         end select
      end do
    end subroutine decay_chain_build_term_entries
```

Build the entries that correspond to a specific decay. The decay term should
have been determined, so we allocate a decay chain entry and fill it, then proceed
to child decays.

For the first entry, we convolute the connected state of the parent process
instance with the isolated state of the current decay (which does not contain an
extra beam entry for the parent). For subsequent entries, we take the previous
entry as first factor.

In principle, each chain entry (as a parton state) is capable of holding a
subevent object and associated expressions. We currently do not make use of
that feature.

Before generating the decays, factor out the trace of the helicity density
matrix of the parent parton state. This trace has been used for unweighting the
original event (unweighted case) or it determines the overall weight, so it should
not be taken into account in the decay chain generation.

⟨*Decays: decay chain: TBP*⟩+≡
```
    procedure :: build_decay_entries => decay_chain_build_decay_entries
```

⟨*Decays: procedures*⟩+≡
```
    recursive subroutine decay_chain_build_decay_entries (chain, decay)
      class(decay_chain_t), intent(inout) :: chain
      type(decay_t), intent(in) :: decay
      type(decay_chain_entry_t), pointer :: entry
      type(connected_state_t), pointer :: previous_state
      type(isolated_state_t), pointer :: current_decay
      allocate (entry)
      if (associated (chain%last)) then
```

```
        entry%previous => chain%last
        entry%index = entry%previous%index + 1
        previous_state => entry%previous%connected_state_t
    else
        entry%index = 1
        previous_state => &
            chain%process_instance%get_connected_state_ptr (chain%selected_term)
    end if
    entry%config => decay%config
    entry%selected_mci = decay%selected_mci
    entry%selected_term = decay%selected_term
    current_decay => decay%process_instance%get_isolated_state_ptr &
        (decay%selected_term)
    call entry%setup_connected_trace &
        (current_decay, previous_state%get_trace_int_ptr ())
    call entry%setup_connected_matrix &
        (current_decay, previous_state%get_matrix_int_ptr ())
    call entry%setup_connected_flows &
        (current_decay, previous_state%get_flows_int_ptr ())
    chain%last => entry
    call chain%build_term_entries (decay%term(decay%selected_term))
  end subroutine decay_chain_build_decay_entries
```

Recursively fill the decay chain with momenta and evaluate the matrix elements. Since all evaluators should have correct source entries at this point, momenta are automatically retrieved from the appropriate process instance.

Like we did above for the parent process, factor out the trace for each subsequent decay (the helicity density matrix in the isolated state, which is taken for the convolution).

⟨*Decays: decay chain: TBP*⟩+≡

```
    procedure :: evaluate => decay_chain_evaluate
```

⟨*Decays: procedures*⟩+≡

```
    subroutine decay_chain_evaluate (chain)
      class(decay_chain_t), intent(inout) :: chain
      call evaluate (chain%last)
      call evaluator_receive_momenta (chain%correlated_trace)
      call chain%correlated_trace%evaluate ()
    contains
      recursive subroutine evaluate (entry)
        type(decay_chain_entry_t), intent(inout), pointer :: entry
        if (associated (entry)) then
           call evaluate (entry%previous)
           call entry%receive_kinematics ()
           call entry%evaluate_trace ()
           call entry%evaluate_event_data ()
        end if
      end subroutine evaluate
    end subroutine decay_chain_evaluate
```

Return the probability of a decay chain. This is given as the trace of the density matrix with intermediate helicity correlations, normalized by the product of the uncorrelated density matrix traces. This works only if an event has been

evaluated and the `correlated_trace` evaluator is filled. By definition, this evaluator has only one matrix element, and this must be real.

⟨*Decays: decay chain: TBP*⟩+≡
```
procedure :: get_probability => decay_chain_get_probability
```

⟨*Decays: procedures*⟩+≡
```
function decay_chain_get_probability (chain) result (x)
  class(decay_chain_t), intent(in) :: chain
  real(default) :: x
  x = evaluator_get_matrix_element (chain%correlated_trace, 1)
end function decay_chain_get_probability
```

17.2.12 Decay as Event Transform

The `evt_decay` object combines decay configuration, decay tree, and chain in a single object, as an implementation of the `evt` (event transform) abstract type.

⟨*Decays: public*⟩≡
```
public :: evt_decay_t
```

⟨*Decays: types*⟩+≡
```
type, extends (evt_t) :: evt_decay_t
  type(decay_root_config_t) :: decay_root_config
  type(decay_root_t) :: decay_root
  type(decay_chain_t) :: decay_chain
 contains
 ⟨Decays: evt decay: TBP⟩
end type evt_decay_t
```

Output. We display the currently selected decay tree, which includes configuration data, and the decay chain, i.e., the evaluators.

⟨*Decays: evt decay: TBP*⟩≡
```
procedure :: write => evt_decay_write
```

⟨*Decays: procedures*⟩+≡
```
subroutine evt_decay_write (object, unit, &
     show_decay_tree, &
     show_processes, &
     verbose)
  class(evt_decay_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_decay_tree, show_processes, verbose
  logical :: dec, prc, verb
  integer :: u
  u = output_unit (unit)
  dec = .true.;  if (present (show_decay_tree))  dec = show_decay_tree
  prc = .false.;  if (present (show_processes))  prc = show_processes
  verb = .false.;  if (present (verbose))  verb = verbose
  call write_separator_double (u)
  write (u, "(1x,A)")  "Event transform: partonic decays"
  call write_separator_double (u)
  call object%base_write (u)
  if (dec) then
     call write_separator (u)
```

2359

```
          call object%decay_root%write (u)
          if (verb) then
             call object%decay_chain%write (u)
          end if
          if (prc) then
             call object%decay_root%write_process_instances (u, verb)
          end if
       else
          call write_separator_double (u)
       end if
    end subroutine evt_decay_write
```

Connect with a process instance and process. This initializes the decay config-
uration. The process stack is used to look for process objects that implement
daughter decays.

When all processes are assigned, configure the decay tree instance, using the
decay tree configuration. First obtain the branching ratios, then allocate the
decay tree. This is done once for all events.

⟨*Decays: evt decay: TBP*⟩+≡
```
    procedure :: connect => evt_decay_connect
```

⟨*Decays: procedures*⟩+≡
```
    subroutine evt_decay_connect (evt, process_instance, model, process_stack)
      class(evt_decay_t), intent(inout), target :: evt
      type(process_instance_t), intent(in), target :: process_instance
      type(model_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      call evt%base_connect (process_instance, model)
      call evt%decay_root_config%connect (process_instance%process, &
           model, process_stack, process_instance)
      call evt%decay_root_config%compute ()
      call evt%decay_root%init (evt%decay_root_config, evt%process_instance)
    end subroutine evt_decay_connect
```

Prepare a new event: Select a decay chain and build the corresponding chain
object.

⟨*Decays: evt decay: TBP*⟩+≡
```
    procedure :: prepare_new_event => evt_decay_prepare_new_event
```

⟨*Decays: procedures*⟩+≡
```
    subroutine evt_decay_prepare_new_event (evt, i_mci, i_term)
      class(evt_decay_t), intent(inout) :: evt
      integer, intent(in) :: i_mci, i_term
      call evt%reset ()
      evt%decay_root%selected_mci = i_mci
      evt%decay_root%selected_term = i_term
      call evt%decay_root%select_chain ()
      call evt%decay_chain%build (evt%decay_root)
    end subroutine evt_decay_prepare_new_event
```

Generate a weighted event and assign the resulting weight (probability). We
use a chain initialized by the preceding subroutine, fill it with momenta and

evaluate.

⟨*Decays: evt decay: TBP*⟩+≡
```
    procedure :: generate_weighted => evt_decay_generate_weighted
```

⟨*Decays: procedures*⟩+≡
```
    subroutine evt_decay_generate_weighted (evt, probability)
      class(evt_decay_t), intent(inout) :: evt
      real(default), intent(out) :: probability
      call evt%decay_root%generate ()
      if (signal_is_pending ())  return
      call evt%decay_chain%evaluate ()
      probability = evt%decay_chain%get_probability ()
    end subroutine evt_decay_generate_weighted
```

To create a usable event, we have to transform the interaction into a particle
set; this requires factorization for the correlated density matrix, according to
the factorization mode.

⟨*Decays: evt decay: TBP*⟩+≡
```
    procedure :: make_particle_set => evt_decay_make_particle_set
```

⟨*Decays: procedures*⟩+≡
```
    subroutine evt_decay_make_particle_set &
        (evt, factorization_mode, keep_correlations, r)
      class(evt_decay_t), intent(inout) :: evt
      integer, intent(in) :: factorization_mode
      logical, intent(in) :: keep_correlations
      real(default), dimension(:), intent(in), optional :: r
      type(interaction_t), pointer :: int_matrix, int_flows
      type(decay_chain_entry_t), pointer :: last_entry
      last_entry => evt%decay_chain%last
      int_matrix => last_entry%get_matrix_int_ptr ()
      int_flows  => last_entry%get_flows_int_ptr ()
      call evt%factorize_interactions (int_matrix, int_flows, &
          factorization_mode, keep_correlations, r)
      call evt%tag_incoming ()
    end subroutine evt_decay_make_particle_set
```

Eliminate numerical noise for the associated process instances.

⟨*Decays: public*⟩+≡
```
    public :: pacify
```

⟨*Decays: interfaces*⟩+≡
```
    interface pacify
       module procedure pacify_decay
    end interface pacify
```

⟨*Decays: procedures*⟩+≡
```
    subroutine pacify_decay (evt)
      class(evt_decay_t), intent(inout) :: evt
      call pacify_decay_gen (evt%decay_root)
    contains
      recursive subroutine pacify_decay_gen (decay)
        class(decay_gen_t), intent(inout) :: decay
        if (associated (decay%process_instance)) then
```

```
            call pacify (decay%process_instance)
         end if
         if (decay%selected_term > 0) then
            call pacify_term (decay%term(decay%selected_term))
         end if
      end subroutine pacify_decay_gen
      recursive subroutine pacify_term (term)
        class(decay_term_t), intent(inout) :: term
        integer :: i
        do i = 1, size (term%particle_out)
           select type (unstable => term%particle_out(i)%c)
           type is (unstable_t);  call pacify_unstable (unstable)
           end select
        end do
      end subroutine pacify_term
      recursive subroutine pacify_unstable (unstable)
        class(unstable_t), intent(inout) :: unstable
        if (unstable%selected_decay > 0) then
           call pacify_decay_gen (unstable%decay(unstable%selected_decay))
        end if
      end subroutine pacify_unstable
    end subroutine pacify_decay
```

17.2.13 Unit tests

⟨*Decays: public*⟩+≡
```
    public :: decays_test
```

⟨*Decays: tests*⟩≡
```
    subroutine decays_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Decays: execute tests*⟩
```
      end subroutine decays_test
```

Testbed

As a variation of the `prepare_test_process` routine used elsewhere, we define
here a routine that creates two processes (scattering $ss \to ss$ and decay $s \to f\bar{f}$),
compiles and integrates them and prepares for event generation.

⟨*Decays: public*⟩+≡
```
    public :: prepare_testbed
```

⟨*Decays: tests*⟩+≡
```
    subroutine prepare_testbed &
         (lib, process_stack, model_list, model, prefix, os_data, &
          scattering, decay, decay_rest_frame)
      type(process_library_t), intent(out), target :: lib
      type(process_stack_t), intent(out) :: process_stack
      type(model_list_t), intent(out) :: model_list
      type(model_t), pointer, intent(out) :: model
      type(string_t), intent(in) :: prefix
```

```
type(os_data_t), intent(in) :: os_data
logical, intent(in) :: scattering, decay
logical, intent(in), optional :: decay_rest_frame
type(string_t) :: libname, procname1, procname2, run_id
type(qcd_t) :: qcd
class(rng_factory_t), allocatable :: rng_factory
type(var_list_t), pointer :: var_list
type(process_entry_t), pointer :: process
type(process_instance_t), allocatable, target :: process_instance
class(prc_core_t), allocatable :: core_template
class(mci_t), allocatable :: mci_template
class(phs_config_t), allocatable :: phs_config_template
type(particle_data_t), pointer :: prt_data
real(default) :: sqrts

libname = prefix // "_lib"
procname1 = prefix // "_p"
procname2 = prefix // "_d"
run_id = prefix

call model_list%read_model (var_str ("Test"), &
     var_str ("Test.mdl"), os_data, model)
var_list => model_get_var_list_ptr (model)
call var_list_set_real (var_list, &
     var_str ("ff"), 0.4_default, is_known = .true.)
call model_parameters_update (model)

if (scattering .and. decay) then
   prt_data => model_get_particle_ptr (model, 25)
   call particle_data_set (prt_data, p_is_stable = .false.)
end if

call prc_test_create_library (libname, lib, &
     scattering = .true., decay = .true., &
     procname1 = procname1, procname2 = procname2)

call reset_interaction_counter ()

allocate (test_t :: core_template)
allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)

if (scattering) then
   allocate (rng_test_factory_t :: rng_factory)
   allocate (process)
   call process%init (procname1, &
        run_id, lib, os_data, qcd, rng_factory, model_list)
   call process%init_component &
        (1, core_template, mci_template, phs_config_template)
   sqrts = 1000
   call process%setup_beams_sqrts (sqrts)
   call process%configure_phs ()
   call process%setup_mci ()
   call process%setup_terms ()
```

```
      allocate (process_instance)
      call process_instance%init (process%process_t)
      call process%integrate (process_instance, 1, n_it=1, n_calls=100)
      call process%final_integration (1)
      call process_instance%final ()
      deallocate (process_instance)

      call process%prepare_simulation (1)
      call process_stack%push (process)
   end if

   if (decay) then
      allocate (rng_test_factory_t :: rng_factory)
      allocate (process)
      call process%init (procname2, &
           run_id, lib, os_data, qcd, rng_factory, model_list)
      call process%init_component &
           (1, core_template, mci_template, phs_config_template)
      if (present (decay_rest_frame)) then
         call process%setup_beams_decay (rest_frame = decay_rest_frame)
      else
         call process%setup_beams_decay (rest_frame = .not. scattering)
      end if
      call process%configure_phs ()
      call process%setup_mci ()
      call process%setup_terms ()

      allocate (process_instance)
      call process_instance%init (process%process_t)
      call process%integrate (process_instance, 1, n_it=1, n_calls=100)
      call process%final_integration (1)
      call process_instance%final ()
      deallocate (process_instance)

      call process%prepare_simulation (1)
      call process_stack%push (process)
   end if

end subroutine prepare_testbed
```

Simple decay configuration

We define a branching configuration with two decay modes. We set the integral
values by hand, so we do not need to evaluate processes, yet.

⟨*Decays: execute tests*⟩≡
```
   call test (decays_1, "decays_1", &
        "branching and decay configuration", &
        u, results)
```

⟨*Decays: tests*⟩+≡
```
   subroutine decays_1 (u)
     integer, intent(in) :: u
```

```
type(os_data_t) :: os_data
type(model_t), pointer :: model
type(model_list_t) :: model_list
type(flavor_t) :: flv_h
type(flavor_t), dimension(2,1) :: flv_hbb, flv_hgg
type(unstable_config_t), allocatable :: unstable

write (u, "(A)")  "* Test output: decays_1"
write (u, "(A)")  "*   Purpose: Set up branching and decay configuration"
write (u, "(A)")

write (u, "(A)")  "* Initialize environment"
write (u, "(A)")

call syntax_model_file_init ()
call os_data_init (os_data)
call model_list%read_model (var_str ("SM"), &
     var_str ("SM.mdl"), os_data, model)

call flavor_init (flv_h, 25, model)
call flavor_init (flv_hbb(:,1), [5, -5], model)
call flavor_init (flv_hgg(:,1), [22, 22], model)

write (u, "(A)")  "* Set up branching and decay"
write (u, "(A)")

allocate (unstable)
unstable%flv = flv_h
call unstable%init_decays ([var_str ("h_bb"), var_str ("h_gg")], model)

associate (decay => unstable%decay_config(1))
  allocate (decay%term_config (1))
  call decay%init_term (1, flv_hbb, stable = [.true., .true.], model=model)
  decay%integral = 1.234e-3_default
  decay%abs_error = decay%integral * .02_default
end associate

associate (decay => unstable%decay_config(2))
  allocate (decay%term_config (1))
  call decay%init_term (1, flv_hgg, stable = [.true., .true.], model=model)
  decay%integral = 3.085e-4_default
  decay%abs_error = decay%integral * .08_default
end associate

call unstable%compute ()
call unstable%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call unstable%final ()
call model_list%final ()
call syntax_model_file_final ()
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Test output end: decays_1"

    end subroutine decays_1
```

Cascade decay configuration

We define a branching configuration with one decay, which is followed by another
branching.

⟨*Decays: execute tests*⟩+≡
```
    call test (decays_2, "decays_2", &
        "cascade decay configuration", &
        u, results)
```

⟨*Decays: tests*⟩+≡
```
    subroutine decays_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_t), pointer :: model
      type(model_list_t) :: model_list
      type(flavor_t) :: flv_h, flv_wp, flv_wm
      type(flavor_t), dimension(2,1) :: flv_hww, flv_wud, flv_wen
      type(unstable_config_t), allocatable :: unstable
      type(string_t), dimension(:), allocatable :: decay

      write (u, "(A)")   "* Test output: decays_2"
      write (u, "(A)")   "*   Purpose: Set up cascade branching"
      write (u, "(A)")

      write (u, "(A)")   "* Initialize environment"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model (var_str ("SM"), &
          var_str ("SM.mdl"), os_data, model)

      call model_set_unstable (model, 25, [var_str ("h_ww")])
      call model_set_unstable (model, 24, [var_str ("w_ud"), var_str ("w_en")])

      call flavor_init (flv_h, 25, model)
      call flavor_init (flv_hww(:,1), [24, -24], model)
      call flavor_init (flv_wp, 24, model)
      call flavor_init (flv_wm,-24, model)
      call flavor_init (flv_wud(:,1), [2, -1], model)
      call flavor_init (flv_wen(:,1), [-11, 12], model)


      write (u, "(A)")   "* Set up branching and decay"
      write (u, "(A)")

      allocate (unstable)
      unstable%flv = flv_h
```

2366

```
call flavor_get_decays (unstable%flv, decay)
call unstable%init_decays (decay, model)

associate (decay => unstable%decay_config(1))

  decay%integral = 1.e-3_default
  decay%abs_error = decay%integral * .01_default

  allocate (decay%term_config (1))
  call decay%init_term (1, flv_hww, stable = [.false., .true.], model=model)

  select type (w => decay%term_config(1)%prt(1)%c)
  type is (unstable_config_t)

     associate (w_decay => w%decay_config(1))
       w_decay%integral = 2._default
       allocate (w_decay%term_config (1))
       call w_decay%init_term (1, flv_wud, stable = [.true., .true.], &
            model=model)
     end associate
     associate (w_decay => w%decay_config(2))
       w_decay%integral = 1._default
       allocate (w_decay%term_config (1))
       call w_decay%init_term (1, flv_wen, stable = [.true., .true.], &
            model=model)
     end associate
     call w%compute ()

  end select

end associate

call unstable%compute ()
call unstable%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call unstable%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: decays_2"

end subroutine decays_2
```

Decay and Process Object

We define a branching configuration with one decay and connect this with an
actual process object.

⟨*Decays: execute tests*⟩+≡

```
      call test (decays_3, "decays_3", &
           "associate process", &
           u, results)
```

⟨*Decays: tests*⟩+≡

```
    subroutine decays_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_library_t), target :: lib
      type(string_t) :: prefix
      type(string_t) :: procname2
      type(process_stack_t) :: process_stack
      type(process_t), pointer :: process
      type(unstable_config_t), allocatable :: unstable

      write (u, "(A)")  "* Test output: decays_3"
      write (u, "(A)")  "*   Purpose: Connect a decay configuration &
           &with a process"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize environment and integrate process"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call syntax_model_file_init ()

      prefix = "decays_3"
      call prepare_testbed &
           (lib, process_stack, model_list, model, prefix, os_data, &
           scattering=.false., decay=.true., decay_rest_frame=.false.)

      procname2 = prefix // "_d"
      process => process_stack%get_process_ptr (procname2)
      call process%write (.false., u)

      write (u, "(A)")
      write (u, "(A)")  "* Set up branching and decay"
      write (u, "(A)")

      allocate (unstable)
      call flavor_init (unstable%flv, 25, model)
      call unstable%init_decays ([procname2], model)

      write (u, "(A)")  "* Connect decay with process object"
      write (u, "(A)")

      associate (decay => unstable%decay_config(1))
        call decay%connect (process, model)
      end associate

      call unstable%compute ()
      call unstable%write (u)
```

2368

```
        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call unstable%final ()
        call process_stack%final ()
        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: decays_3"

    end subroutine decays_3
```

Decay and Process Object

Building upon the previous test, we set up a decay instance and generate a decay event.

⟨*Decays: execute tests*⟩+≡
```
    call test (decays_4, "decays_4", &
         "decay instance", &
         u, results)
```

⟨*Decays: tests*⟩+≡
```
    subroutine decays_4 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_library_t), target :: lib
      type(string_t) :: prefix, procname2
      class(rng_t), allocatable :: rng
      type(process_stack_t) :: process_stack
      type(process_t), pointer :: process
      type(unstable_config_t), allocatable, target :: unstable
      type(unstable_t), allocatable :: instance

      write (u, "(A)")  "* Test output: decays_4"
      write (u, "(A)")  "*   Purpose: Create a decay process and evaluate &
           &an instance"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize environment, process, &
           &and decay configuration"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call syntax_model_file_init ()

      prefix = "decays_4"
      call prepare_testbed &
           (lib, process_stack, model_list, model, prefix, os_data, &
```

2369

```
         scattering=.false., decay=.true., decay_rest_frame = .false.)

    procname2 = prefix // "_d"
    process => process_stack%get_process_ptr (procname2)

    allocate (unstable)
    call flavor_init (unstable%flv, 25, model)
    call unstable%init_decays ([procname2], model)

    call model_set_unstable (model, 25, [procname2])

    associate (decay => unstable%decay_config(1))
      call decay%connect (process, model)
    end associate

    call unstable%compute ()

    allocate (rng_test_t :: rng)

    allocate (instance)
    call instance%init (unstable)
    call instance%import_rng (rng)

    call instance%select_chain ()
    call instance%generate ()
    call instance%write (u)

    write (u, *)
    call instance%write_process_instances (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call instance%final ()
    call process_stack%final ()
    call unstable%final ()
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: decays_4"

  end subroutine decays_4
```

Decay with Parent Process

We define a scattering process $ss \rightarrow ss$ and subsequent decays $s \rightarrow f\bar{f}$.

⟨*Decays: execute tests*⟩+≡
```
    call test (decays_5, "decays_5", &
         "parent process and decay", &
         u, results)
```

⟨*Decays: tests*⟩+≡

```
subroutine decays_5 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(process_library_t), target :: lib
  type(string_t) :: prefix, procname1, procname2
  type(process_stack_t) :: process_stack
  type(process_t), pointer :: process
  type(process_instance_t), allocatable, target :: process_instance
  type(decay_root_config_t), target :: decay_root_config
  type(decay_root_t) :: decay_root
  type(decay_chain_t) :: decay_chain
  integer :: i

  write (u, "(A)")  "* Test output: decays_5"
  write (u, "(A)")  "*   Purpose: Handle a process with subsequent decays"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize environment and parent process"
  write (u, "(A)")

  call syntax_model_file_init ()
  call os_data_init (os_data)
  call syntax_model_file_init ()

  prefix = "decays_5"
  procname1 = prefix // "_p"
  procname2 = prefix // "_d"
  call prepare_testbed &
       (lib, process_stack, model_list, model, prefix, os_data, &
       scattering=.true., decay=.true.)

  write (u, "(A)")  "* Initialize decay process"
  write (u, "(A)")

  call model_set_unstable (model, 25, [procname2])

  write (u, "(A)")  "* Initialize decay tree configuration"
  write (u, "(A)")

  process => process_stack%get_process_ptr (procname1)
  call decay_root_config%connect (process, model, process_stack)
  call decay_root_config%compute ()
  call decay_root_config%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize decay tree"

  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data ()
  call process_instance%init_simulation (1)
```

2371

```
call decay_root%init (decay_root_config, process_instance)

write (u, "(A)")
write (u, "(A)")  "* Select decay chain"
write (u, "(A)")

decay_root%selected_mci = 1
!!! Not yet implemented; there is only one term anyway:
! call process_instance%select_i_term (decay_root%selected_term)
decay_root%selected_term = 1
call decay_root%select_chain ()

call decay_chain%build (decay_root)

call decay_root%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate event"
write (u, "(A)")

call process%generate_unweighted_event (process_instance, &
     decay_root%selected_mci)
call process_instance%evaluate_event_data ()

call decay_root%generate ()

associate (term => decay_root%term(1))
  do i = 1, 2
     select type (unstable => term%particle_out(i)%c)
     type is (unstable_t)
        associate (decay => unstable%decay(1))
          call pacify (decay%process_instance)
        end associate
     end select
  end do
end associate

write (u, "(A)")  "* Process instances"
write (u, "(A)")

call decay_root%write_process_instances (u)

write (u, "(A)")
write (u, "(A)")  "* Generate decay chain"
write (u, "(A)")

call decay_chain%evaluate ()
call decay_chain%write (u)

write (u, *)
write (u, "(A,ES19.12)")  "chain probability =", &
     decay_chain%get_probability ()

write (u, "(A)")
```

2372

```
      write (u, "(A)")   "* Cleanup"

      call decay_chain%final ()
      call decay_root%final ()
      call decay_root_config%final ()
      call process_instance%final ()
      deallocate (process_instance)

      call process%final ()
      call process_stack%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: decays_5"

    end subroutine decays_5
```

Decay as Event Transform

Again, we define a scattering process $ss \to ss$ and subsequent decays $s \to f\bar{f}$.

⟨*Decays: execute tests*⟩+≡
```
    call test (decays_6, "decays_6", &
         "evt_decay object", &
         u, results)
```

⟨*Decays: tests*⟩+≡
```
    subroutine decays_6 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_library_t), target :: lib
      type(string_t) :: prefix, procname1, procname2
      type(process_stack_t) :: process_stack
      type(process_t), pointer :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(evt_decay_t), target :: evt_decay
      integer :: factorization_mode
      logical :: keep_correlations

      write (u, "(A)")   "* Test output: decays_6"
      write (u, "(A)")   "*   Purpose: Handle a process with subsequent decays"
      write (u, "(A)")

      write (u, "(A)")   "* Initialize environment and parent process"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)

      prefix = "decays_6"
      procname1 = prefix // "_p"
```

2373

```
      procname2 = prefix // "_d"
      call prepare_testbed &
           (lib, process_stack, model_list, model, prefix, os_data, &
           scattering=.true., decay=.true.)

      write (u, "(A)")  "* Initialize decay process"

      call model_set_unstable (model, 25, [procname2])

      process => process_stack%get_process_ptr (procname1)
      allocate (process_instance)
      call process_instance%init (process)
      call process_instance%setup_event_data ()
      call process_instance%init_simulation (1)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize decay object"

      call evt_decay%connect (process_instance, model, process_stack)

      write (u, "(A)")
      write (u, "(A)")  "* Generate scattering event"

      call process%generate_unweighted_event (process_instance, 1)
      call process_instance%evaluate_event_data ()

      write (u, "(A)")
      write (u, "(A)")  "* Select decay chain and generate event"
      write (u, "(A)")

      call evt_decay%prepare_new_event (1, 1)
      call evt_decay%generate_unweighted ()

      factorization_mode = FM_IGNORE_HELICITY
      keep_correlations = .false.
      call evt_decay%make_particle_set (factorization_mode, keep_correlations)

      call evt_decay%write (u, verbose = .true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call evt_decay%final ()
      call process_instance%final ()
      deallocate (process_instance)

      call process_stack%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: decays_6"

    end subroutine decays_6
```

17.3 Shower

⟨shower.f90⟩≡
  ⟨*File header*⟩

  module shower

    use kinds, only: default, double !NODEP!
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use constants, only : pi, twopi !NODEP!
    use limits, only: LF !NODEP!
    use diagnostics !NODEP!
    use lorentz !NODEP!
    use shower_base !NODEP!
    use shower_partons !NODEP!
    use shower_core !NODEP!
    use shower_topythia !NODEP!
    use muli, muli_output_unit => output_unit !NODEP!
    use mlm_matching !NODEP!
    use ckkw_pseudo_weights !NODEP!
    use ckkw_matching !NODEP!
    use tao_random_numbers !NODEP!
    use pdf_builtin !NODEP!

    use unit_tests
    use os_interface
    use xml

    use sm_qcd
    use flavors
    use colors
    use particles
    use state_matrices
    use subevents
    use models
    use variables
    use beam_structures
    use hep_common
    use process_libraries
    use prc_core
    use prc_omega
    use rng_base
    use rng_tao
    use mci_base
    use mci_midpoint
    use phs_base
    use phs_single
    use processes
    use event_transforms

⟨*Standard module head*⟩

⟨*Shower: public*⟩

⟨*Shower: parameters*⟩

⟨*Shower: types*⟩

```
contains
```

⟨*Shower: procedures*⟩

⟨*Shower: tests*⟩

```
end module shower
```

17.3.1 PDF Parameters

The shower has to know about the PDF set used for the parent process, if any.
We support the following:

⟨*Shower: parameters*⟩≡
```
    integer, parameter :: STRF_NONE = 0
    integer, parameter :: STRF_LHAPDF = 1
    integer, parameter :: STRF_PDF_BUILTIN = 2
```

17.3.2 Configuration Parameters

⟨*Shower: public*⟩≡
```
    public :: shower_settings_t
```

⟨*Shower: types*⟩≡
```
    type :: shower_settings_t
       logical :: ps_isr_active = .false.
       logical :: ps_fsr_active = .false.
       logical :: ps_use_PYTHIA_shower = .false.
       logical :: hadronization_active = .false.
       logical :: mlm_matching = .false.
       logical :: ckkw_matching = .false.
       logical :: muli_active = .false.

       logical :: ps_PYTHIA_verbose = .false.
       type(string_t) :: ps_PYTHIA_PYGIVE

       !!! values present in PYTHIA and WHIZARDs PS,
       !!! comments denote corresponding PYTHIA values
       real(default) :: ps_mass_cutoff = 1._default       ! PARJ(82)
       real(default) :: ps_fsr_lambda = 0.29_default      ! PARP(72)
       real(default) :: ps_isr_lambda = 0.29_default      ! PARP(61)
       integer :: ps_max_n_flavors = 5                    ! MSTJ(45)
       logical :: ps_isr_alpha_s_running = .true.         ! MSTP(64)
       logical :: ps_fsr_alpha_s_running = .true.         ! MSTJ(44)
       real(default) :: ps_fixed_alpha_s = 0._default     ! PARU(111)
```

```
      logical :: ps_isr_pt_ordered = .false.
      logical :: ps_isr_angular_ordered = .true.         ! MSTP(62)
      real(default) :: ps_isr_primordial_kt_width = 0._default   ! PARP(91)
      real(default) :: ps_isr_primordial_kt_cutoff = 5._default ! PARP(93)
      real(default) :: ps_isr_z_cutoff = 0.999_default  ! 1-PARP(66)
      real(default) :: ps_isr_minenergy = 2._default     ! PARP(65)
      real(default) :: ps_isr_tscalefactor = 1._default
      logical :: ps_isr_only_onshell_emitted_partons = .true.   ! MSTP(63)

      !!! MLM settings
      type(mlm_matching_settings_t) :: ms

      !!! CKKW Matching
      type(ckkw_matching_settings_t) :: ckkw_settings
      type(ckkw_pseudo_shower_weights_t) :: ckkw_weights
    contains
    ⟨Shower: shower settings: TBP⟩
    end type shower_settings_t
```

Read in the shower, matching and hadronization settings

⟨*Shower: shower settings: TBP*⟩≡
```
    procedure :: init => shower_settings_init
```

⟨*Shower: procedures*⟩≡
```
    subroutine shower_settings_init (shower_settings, var_list)
      class(shower_settings_t), intent(out) :: shower_settings
      type(var_list_t), intent(in) :: var_list

      shower_settings%ps_isr_active = &
           var_list_get_lval (var_list, var_str ("?ps_isr_active"))
      shower_settings%ps_fsr_active = &
           var_list_get_lval (var_list, var_str ("?ps_fsr_active"))
      shower_settings%hadronization_active = &
           var_list_get_lval (var_list, var_str ("?hadronization_active"))
      shower_settings%mlm_matching = &
           var_list_get_lval (var_list, var_str ("?mlm_matching"))
      shower_settings%ckkw_matching = &
           var_list_get_lval (var_list, var_str ("?ckkw_matching"))
      shower_settings%muli_active = &
           var_list_get_lval (var_list, var_str ("?muli_active"))

      if (.not. shower_settings%ps_fsr_active .and. &
          .not. shower_settings%ps_isr_active .and. &
          .not. shower_settings%hadronization_active .and. &
          .not. shower_settings%mlm_matching) then
        return
      end if

      shower_settings%ps_use_PYTHIA_shower = &
           var_list_get_lval (var_list, var_str ("?ps_use_PYTHIA_shower"))
      shower_settings%ps_PYTHIA_verbose = &
           var_list_get_lval (var_list, var_str ("?ps_PYTHIA_verbose"))
      shower_settings%ps_PYTHIA_PYGIVE = &
           var_list_get_sval (var_list, var_str ("$ps_PYTHIA_PYGIVE"))
```

```
shower_settings%ps_mass_cutoff = &
     var_list_get_rval (var_list, var_str ("ps_mass_cutoff"))
shower_settings%ps_fsr_lambda = &
     var_list_get_rval (var_list, var_str ("ps_fsr_lambda"))
shower_settings%ps_isr_lambda = &
     var_list_get_rval (var_list, var_str ("ps_isr_lambda"))
shower_settings%ps_max_n_flavors = &
     var_list_get_ival (var_list, var_str ("ps_max_n_flavors"))
shower_settings%ps_isr_alpha_s_running = &
     var_list_get_lval (var_list, var_str ("?ps_isr_alpha_s_running"))
shower_settings%ps_fsr_alpha_s_running = &
     var_list_get_lval (var_list, var_str ("?ps_fsr_alpha_s_running"))
shower_settings%ps_fixed_alpha_s = &
     var_list_get_rval (var_list, var_str ("ps_fixed_alpha_s"))
shower_settings%ps_isr_pt_ordered = &
     var_list_get_lval (var_list, var_str ("?ps_isr_pt_ordered"))
shower_settings%ps_isr_angular_ordered = &
     var_list_get_lval (var_list, var_str ("?ps_isr_angular_ordered"))
shower_settings%ps_isr_primordial_kt_width = &
     var_list_get_rval (var_list, var_str ("ps_isr_primordial_kt_width"))
shower_settings%ps_isr_primordial_kt_cutoff = &
     var_list_get_rval (var_list, var_str ("ps_isr_primordial_kt_cutoff"))
shower_settings%ps_isr_z_cutoff = &
     var_list_get_rval (var_list, var_str ("ps_isr_z_cutoff"))
shower_settings%ps_isr_minenergy = &
     var_list_get_rval (var_list, var_str ("ps_isr_minenergy"))
shower_settings%ps_isr_tscalefactor = &
     var_list_get_rval (var_list, var_str ("ps_isr_tscalefactor"))
shower_settings%ps_isr_only_onshell_emitted_partons = &
     var_list_get_lval (var_list, &
     var_str ("?ps_isr_only_onshell_emitted_partons"))

!!! MLM matching
shower_settings%ms%mlm_Qcut_ME = &
     var_list_get_rval(var_list, var_str ("mlm_Qcut_ME"))
shower_settings%ms%mlm_Qcut_PS = &
     var_list_get_rval(var_list, var_str ("mlm_Qcut_PS"))
shower_settings%ms%mlm_ptmin = &
     var_list_get_rval(var_list, var_str ("mlm_ptmin"))
shower_settings%ms%mlm_etamax = &
     var_list_get_rval(var_list, var_str ("mlm_etamax"))
shower_settings%ms%mlm_Rmin = &
     var_list_get_rval(var_list, var_str ("mlm_Rmin"))
shower_settings%ms%mlm_Emin = &
     var_list_get_rval(var_list, var_str ("mlm_Emin"))
shower_settings%ms%mlm_nmaxMEjets = &
     var_list_get_ival(var_list, var_str ("mlm_nmaxMEjets"))

shower_settings%ms%mlm_ETclusfactor = &
     var_list_get_rval(var_list, var_str ("mlm_ETclusfactor"))
shower_settings%ms%mlm_ETclusminE = &
     var_list_get_rval(var_list, var_str ("mlm_ETclusminE"))
shower_settings%ms%mlm_etaclusfactor = &
     var_list_get_rval(var_list, var_str ("mlm_etaclusfactor"))
```

```
        shower_settings%ms%mlm_Rclusfactor = &
             var_list_get_rval(var_list, var_str ("mlm_Rclusfactor"))
        shower_settings%ms%mlm_Eclusfactor = &
             var_list_get_rval(var_list, var_str ("mlm_Eclusfactor"))

        !!! CKKW matching
        ! TODO
     end subroutine shower_settings_init
```

⟨*Shower: shower settings: TBP*⟩+≡
```
     procedure :: write => shower_settings_write
```

⟨*Shower: procedures*⟩+≡
```
     subroutine shower_settings_write (object, unit)
       class(shower_settings_t), intent(in) :: object
       integer, intent(in), optional :: unit
       integer :: u
       u = output_unit (unit);  if (u < 0)  return
       write (u, "(1x,A)")  "Shower settings:"
       call write_separator (u)
       write (u, "(1x,A)")  "Master switches:"
       write (u, "(3x,A,1x,L1)") &
            "ps_isr_active              = ", object%ps_isr_active
       write (u, "(3x,A,1x,L1)") &
            "ps_fsr_active              = ", object%ps_fsr_active
       write (u, "(1x,A)")  "General settings:"
       if (object%ps_isr_active .or. object%ps_fsr_active) then
          write (u, "(3x,A,1x,L1)") &
               "ps_use_PYTHIA_shower       = ", object%ps_use_PYTHIA_shower
          write (u, "(3x,A,ES19.12)") &
               "ps_mass_cutoff             = ", object%ps_mass_cutoff
          write (u, "(3x,A,1x,I1)") &
               "ps_max_n_flavors           = ", object%ps_max_n_flavors
       else
          write (u, "(3x,A)") " [ISR and FSR off]"
       end if
       if (object%ps_isr_active) then
          write (u, "(1x,A)")  "ISR settings:"
          write (u, "(3x,A,1x,L1)") &
               "ps_isr_pt_ordered          = ", object%ps_isr_pt_ordered
          write (u, "(3x,A,ES19.12)") &
               "ps_isr_lambda              = ", object%ps_isr_lambda
          write (u, "(3x,A,1x,L1)") &
               "ps_isr_alpha_s_running     = ", object%ps_isr_alpha_s_running
          write (u, "(3x,A,ES19.12)") &
               "ps_isr_primordial_kt_width = ", object%ps_isr_primordial_kt_width
          write (u, "(3x,A,ES19.12)") &
               "ps_isr_primordial_kt_cutoff = ", &
               object%ps_isr_primordial_kt_cutoff
          write (u, "(3x,A,ES19.12)") &
               "ps_isr_z_cutoff            = ", object%ps_isr_z_cutoff
          write (u, "(3x,A,ES19.12)") &
               "ps_isr_minenergy           = ", object%ps_isr_minenergy
          write (u, "(3x,A,ES19.12)") &
```

```
         "ps_isr_tscalefactor          = ", object%ps_isr_tscalefactor
     else if (object%ps_fsr_active) then
        write (u, "(3x,A)") " [ISR off]"
     end if
     if (object%ps_fsr_active) then
        write (u, "(1x,A)")  "FSR settings:"
        write (u, "(3x,A,ES19.12)") &
             "ps_fsr_lambda             = ", object%ps_fsr_lambda
        write (u, "(3x,A,1x,L1)") &
             "ps_fsr_alpha_s_running    = ", object%ps_fsr_alpha_s_running
     else if (object%ps_isr_active) then
        write (u, "(3x,A)") " [FSR off]"
     end if
     write (u, "(1x,A)")  "Hadronization settings:"
     write (u, "(3x,A,1x,L1)") &
          "hadronization_active        = ", object%hadronization_active
     write (u, "(1x,A)")  "Matching Settings:"
     write (u, "(3x,A,1x,L1)") &
          "mlm_matching                = ", object%mlm_matching
     if (object%mlm_matching) then
        call mlm_matching_settings_write (object%ms, u)
     end if
     write (u, "(3x,A,1x,L1)") &
          "ckkw_matching               = ", object%ckkw_matching
     if (object%ckkw_matching) then
        ! TODO ckkw settings etc.
     end if
     write (u, "(1x,A)")  "PYTHIA specific settings:"
     write (u, "(3x,A,1x,L1)") &
          "ps_PYTHIA_verbose           = ", object%ps_PYTHIA_verbose
     write (u, "(3x,A,A,A)") &
          "ps_PYTHIA_PYGIVE            = '", &
          char(object%ps_PYTHIA_PYGIVE), "'"
   end subroutine shower_settings_write
```

17.3.3 Shower Application Interface

The wrapper subroutine for the shower and the hadronization interface and
MLM matching. This should be the only subroutine called from WHIZARD.

⟨*Shower: public*⟩+≡
```
    public :: apply_shower_particle_set
```

⟨*Shower: procedures*⟩+≡
```
    subroutine apply_shower_particle_set &
         (particle_set, shower_settings,  model, &
          os_data, pdf_type, pdf_set, valid, vetoed)
      type(particle_set_t), intent(inout) :: particle_set
      type(shower_settings_t), intent(in) :: shower_settings
      type(model_t), pointer, intent(in) :: model
      type(os_data_t), intent(in) :: os_data
      integer, intent(in) :: pdf_type
      integer, intent(in) :: pdf_set
      logical, intent(inout) :: valid
```

```
logical, intent(inout) :: vetoed
real(kind=double) :: pdftest
logical, parameter :: debug = .false., to_file = .false.

type(mlm_matching_data_t) :: mlm_matching_data
logical, save :: matching_disabled=.false.
procedure(shower_pdf), pointer :: pdf_func => null()

interface
   subroutine evolvePDFM (set, x, q, ff)
     integer, intent(in) :: set
     double precision, intent(in) :: x, q
     double precision, dimension(-6:6), intent(out) :: ff
   end subroutine evolvePDFM
end interface

if (.not. shower_settings%ps_fsr_active .and. &
    .not. shower_settings%ps_isr_active .and. &
    .not. shower_settings%hadronization_active .and. &
    .not. shower_settings%mlm_matching) then
   ! return if nothing to do
   return
end if

! return if already invalid or vetoed
if (.not. valid .or. vetoed)  return

if (signal_is_pending ()) return

! ensure that lhapdf is initialized
if (pdf_type .eq. STRF_LHAPDF) then
   if (shower_settings%ps_isr_active .and. &
       (abs (particle_get_pdg (particle_set_get_particle &
           (particle_set, 1))) >= 1000) .and. &
       (abs (particle_get_pdg (particle_set_get_particle &
           (particle_set, 2))) >= 1000)) then
      call GetQ2max (0,pdftest)
      if (pdftest == 0._double) then
         call msg_fatal ("ISR QCD shower enabled, but LHAPDF not" // &
              "initialized," // LF // "     aborting simulation")
         return
      end if
   end if
   pdf_func => evolvePDFM
else if (pdf_type == STRF_PDF_BUILTIN) then
   if (shower_settings%ps_use_PYTHIA_shower) then
      call msg_fatal ("Builtin PDFs cannot be used for PYTHIA showers," &
           // LF // "     aborting simulation")
      return
   end if
   pdf_func => pdf_evolve_LHAPDF
end if
if (shower_settings%mlm_matching .and. shower_settings%ckkw_matching) then
   call msg_fatal ("Both MLM and CKKW matching activated," // &
```

```
         LF // "      aborting simulation")
   return
end if

if (debug)  call shower_settings%write ()

if (shower_settings%ps_use_PYTHIA_shower .or. &
     shower_settings%hadronization_active) then
   if (.not. shower_settings%ps_PYTHIA_verbose) then
      call PYGIVE ('MSTU(12)=12345')
      call PYGIVE ('MSTU(13)=0')
   else
      call PYGIVE ('MSTU(13)=1')
   end if
end if
if (debug)  print *, "Shower: beam checks"

if (.not. matching_disabled) then
   !!! Check if the beams are hadrons
   if ((abs (particle_get_pdg (particle_set_get_particle &
        (particle_set, 1))) <= 18) .and. &
        (abs (particle_get_pdg (particle_set_get_particle &
        (particle_set, 2))) <= 18)) then
      mlm_matching_data%is_hadron_collision = .false.
   else if ((abs (particle_get_pdg (particle_set_get_particle &
        (particle_set, 1))) >= 1000) .and. &
        (abs (particle_get_pdg (particle_set_get_particle &
        (particle_set, 2))) >= 1000)) then
      mlm_matching_data%is_hadron_collision = .true.
   else
      call msg_error (" Matching didn't recognize beams setup," // &
           LF // "      disabling matching")
      matching_disabled = .true.
      return
   end if
end if

!!! SHOWER
if (shower_settings%ps_use_PYTHIA_shower .or. &
     (.not. shower_settings%ps_fsr_active .and. &
      .not. shower_settings%ps_isr_active .and. &
       shower_settings%hadronization_active)) then
   call apply_PYTHIAshower_particle_set (particle_set, &
        shower_settings, mlm_matching_data%P_ME, model, valid)
   if (debug)  call pylist(2)
else
   call apply_WHIZARDshower_particle_set (particle_set, &
        shower_settings, mlm_matching_data%P_ME, model, &
        os_data, pdf_func, pdf_set, valid, vetoed)
        if (vetoed) return
end if
if (debug) then
   call particle_set_write (particle_set)
   print *, " after SHOWER"
```

```
      end if

      if (shower_settings%mlm_matching .and. &
          (matching_disabled.eqv..false.)) then
!!! MLM stage 2 -> PS jets and momenta
         call matching_transfer_PS &
              (mlm_matching_data, particle_set, shower_settings)
!!! MLM stage 3 -> reconstruct and possible reject
         call mlm_matching_apply (mlm_matching_data, shower_settings%ms, vetoed)
         if (vetoed) then
            call mlm_matching_data_final (mlm_matching_data)
            return
         end if
      endif

!!! HADRONIZATION
      if (shower_settings%hadronization_active) then
         !! Assume that the event record is still in the PYTHIA COMMON BLOCKS
         !! transferred there by one of the shower routines
         if (valid) then
            call apply_PYTHIAhadronization (particle_set, &
                 shower_settings, model, valid)
         end if
      end if
!!! FINAL

      call mlm_matching_data_final (mlm_matching_data)

      if (debug)  print *, "SHOWER+HAD+MATCHING finished"

    contains

      subroutine shower_set_PYTHIA_error (mstu23)
        ! PYTHIA common blocks
        IMPLICIT DOUBLE PRECISION(A-H, O-Z)
        IMPLICIT INTEGER(I-N)
        COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
        SAVE/PYDAT1/

        integer, intent(in) :: mstu23

        MSTU(23) = mstu23
      end subroutine shower_set_PYTHIA_error

      function shower_get_PYTHIA_error () result (mstu23)
        ! PYTHIA common blocks
        IMPLICIT DOUBLE PRECISION(A-H, O-Z)
        IMPLICIT INTEGER(I-N)
        COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
        SAVE/PYDAT1/

        integer :: mstu23

        mstu23 = MSTU(23)
```

2383

```
end function shower_get_PYTHIA_error

subroutine apply_PYTHIAshower_particle_set &
     (particle_set, shower_settings, JETS_ME, model, valid)
```
⟨*HEP common: parameters*⟩
⟨*HEP common: variables*⟩
⟨*HEP common: common blocks*⟩
```
  type(particle_set_t), intent(inout) :: particle_set
  type(particle_set_t) :: pset_reduced
  type(shower_settings_t), intent(in) :: shower_settings
  type(vector4_t), dimension(:), allocatable, intent(inout) :: JETS_ME
  type(model_t), pointer, intent(in) :: model
  logical, intent(inout) :: valid
  real(kind=default) :: rand

  ! units for transfer from WHIZARD to PYTHIA and back
  integer :: u_W2P, u_P2W
  integer, save :: pythia_initialized_for_NPRUP = 0
  logical, save :: pythia_warning_given = .false.
  logical, save :: msg_written = .false.
  type(string_t) :: remaining_PYGIVE, partial_PYGIVE
  character*10 buffer

  if (signal_is_pending ()) return

  if (debug) then
     print *, "debugging the shower"
     print *, IDBMUP(1), IDBMUP(2)
     print *, EBMUP, PDFGUP, PDFSUP, IDWTUP
     print *, "NPRUP = ", NPRUP
  end if

  ! check if the beam particles are quarks
  if (any (abs(IDBMUP) <= 8)) then
     ! PYTHIA doesn't support these settings
     if (.not. pythia_warning_given) then
        call msg_error ("PYTHIA doesn't support quarks as beam particles," &
             // LF // "      neglecting ISR, FSR and hadronization")
        pythia_warning_given = .true.
     end if
     return
  end if

  call particle_set_reduce (particle_set, pset_reduced)
  call hepeup_from_particle_set (pset_reduced)
  call hepeup_set_event_parameters (proc_id=1)

  u_W2P = free_unit ()
  if (debug .and. to_file) then
     open (unit=u_W2P, status="replace", &
           file="whizardout1.lhe", action="readwrite")
  else
     open (unit=u_W2P, status="scratch", action="readwrite")
```

2384

```
end if
call shower_W2P_write_event (u_W2P)
rewind (u_W2P)
if (signal_is_pending ()) return
write (buffer, "(I10)")  u_W2P
call PYGIVE ("MSTP(161)="//buffer)
call PYGIVE ("MSTP(162)="//buffer)
if (debug)  write (*, "(A)")  buffer
if (shower_settings%ps_isr_active) then
   call PYGIVE ("MSTP(61)=1")
else
   call PYGIVE ("MSTP(61)=0")  !!! switch off ISR
end if
if (shower_settings%ps_fsr_active) then
   call PYGIVE ("MSTP(71)=1")
else
   call PYGIVE ("MSTP(71)=0")   !!! switch off FSR
end if
call PYGIVE ("MSTP(111)=0")     !!! switch off hadronization

if (pythia_initialized_for_NPRUP >= NPRUP) then
   if (debug)  print *, "calling upinit"
   call upinit
   if (debug)  print *, "returned from upinit"
else
   write (buffer, "(F10.5)") shower_settings%ps_mass_cutoff
   call PYGIVE ("PARJ(82)="//buffer)
   write (buffer, "(F10.5)") shower_settings%ps_isr_tscalefactor
   call PYGIVE ("PARP(71)="//buffer)

   write (buffer, "(F10.5)") shower_settings%ps_fsr_lambda
   call PYGIVE ("PARP(72)="//buffer)
   write(buffer, "(F10.5)") shower_settings%ps_isr_lambda
   call PYGIVE ("PARP(61)="//buffer)
   write (buffer, "(I10)") shower_settings%ps_max_n_flavors
   call PYGIVE ("MSTJ(45)="//buffer)
   if (shower_settings%ps_isr_alpha_s_running) then
      call PYGIVE ("MSTP(64)=2")
   else
      call PYGIVE ("MSTP(64)=0")
   end if
   if (shower_settings%ps_fsr_alpha_s_running) then
      call PYGIVE ("MSTJ(44)=2")
   else
      call PYGIVE ("MSTJ(44)=0")
   end if
   write (buffer, "(F10.5)") shower_settings%ps_fixed_alpha_s
   call PYGIVE ("PARU(111)="//buffer)
   write (buffer, "(F10.5)") shower_settings%ps_isr_primordial_kt_width
   call PYGIVE ("PARP(91)="//buffer)
   write (buffer, "(F10.5)") shower_settings%ps_isr_primordial_kt_cutoff
   call PYGIVE ("PARP(93)="//buffer)
   write (buffer, "(F10.5)") 1._double - shower_settings%ps_isr_z_cutoff
   call PYGIVE ("PARP(66)="//buffer)
```

2385

```fortran
      write (buffer, "(F10.5)") shower_settings%ps_isr_minenergy
      call PYGIVE ("PARP(65)="//buffer)
      if (shower_settings%ps_isr_only_onshell_emitted_partons) then
         call PYGIVE ("MSTP(63)=0")
      else
         call PYGIVE ("MSTP(63)=2")
      end if
      if (shower_settings%mlm_matching) then
         CALL PYGIVE ('MSTP(62)=2')
         CALL PYGIVE ('MSTP(67)=0')
      end if
      if (debug)  print *, "calling pyinit"
      call PYINIT ("USER", "", "", 0D0)

      call tao_random_number (rand)
      write (buffer, "(I10)") floor (rand*900000000)
      call pygive ("MRPY(1)="//buffer)
      call pygive ("MRPY(2)=0")

      if (len(shower_settings%ps_PYTHIA_PYGIVE) > 0) then
         remaining_PYGIVE = shower_settings%ps_PYTHIA_PYGIVE
         do while (len (remaining_PYGIVE)>0)
            call split (remaining_PYGIVE, partial_PYGIVE, ";")
            call PYGIVE (char (partial_PYGIVE))
         end do
         if (shower_get_PYTHIA_error() /= 0) then
            call msg_fatal &
                  (" PYTHIA did not recognize ps_PYTHIA_PYGIVE setting.")
         end if
      end if

      pythia_initialized_for_NPRUP = NPRUP
   end if

   if (.not. msg_written) then
      call msg_message ("Using PYTHIA interface for parton showers")
      msg_written = .true.
   end if
   call PYEVNT ()

   if (debug)  write (*, "(A)")  "called pyevnt"

   u_P2W = free_unit ()
   write (buffer, "(I10)")  u_P2W
   call PYGIVE ("MSTP(163)="//buffer)
   if (debug .and. to_file) then
      open (unit = u_P2W, file="pythiaout.lhe", status="replace", &
            action="readwrite")
   else
      open (unit = u_P2W, status="scratch", action="readwrite")
   end if
   if (debug)  write (*, "(A)")  "calling PYLHEO"
   !!! convert pythia /PYJETS/ to lhef given in MSTU(163)=u_P2W
   call PYLHEO
```

```
      !!! read and add lhef from u_P2W
      if (signal_is_pending ()) return
      call shower_add_lhef_to_particle_set &
            (particle_set, u_P2W, model, os_data)
      close (unit=u_P2W)

      !!! Transfer momenta of the partons in the final state of
      !!!     the hard initeraction
      if (shower_settings%mlm_matching)  &
            call get_ME_momenta_from_PYTHIA (JETS_ME)

      if (shower_get_PYTHIA_error () > 0) then
         !!! clean up, discard shower and exit
         call shower_set_PYTHIA_error (0)
         valid = .false.
      end if
      close (unit=u_W2P)
end subroutine apply_PYTHIAshower_particle_set

subroutine apply_WHIZARDshower_particle_set &
        (particle_set, shower_settings, JETS_ME, model_in, &
        os_data, pdf_func, pdf_set, valid, vetoed)
    type(particle_set_t), intent(inout) :: particle_set
    type(shower_settings_t), intent(in) :: shower_settings
    type(vector4_t), dimension(:), allocatable, intent(inout) :: JETS_ME
    type(model_t), pointer, intent(in) :: model_in
    type(os_data_t), intent(in) :: os_data
    procedure(shower_pdf), pointer, intent(in) :: pdf_func
    integer, intent(in) :: pdf_set
    logical, intent(inout) :: valid
    logical, intent(out) :: vetoed

    type(muli_type),save :: mi
    type(shower_t) :: shower
    type(parton_t), dimension(:), allocatable, target :: partons, hadrons
    type(parton_pointer_t), dimension(:), allocatable :: &
          parton_pointers, final_ME_partons
    real(kind=default) :: mi_scale, ps_scale, shat, phi
    type(parton_pointer_t) :: temppp
    integer, dimension(:), allocatable :: connections
    integer :: n_loop, i, j, k
    integer :: n_hadrons, n_in, n_out
    integer :: n_int
    integer :: max_color_nr
    integer, dimension(2) :: col_array
    integer, dimension(1) :: parent
    type(flavor_t) :: flv
    type(color_t) :: col
    type(model_t), pointer :: model
    type(model_t), target, save :: model_SM_hadrons
    logical, save :: model_SM_hadrons_associated = .false.
    logical, save :: msg_written = .false.
    logical :: exist_SM_hadrons
    type(string_t) :: filename
```

```
      integer, dimension(2,4) :: color_corr
      integer :: colori, colorj
      character*5 buffer
      integer :: u_S2W

      vetoed = .false.

      if (signal_is_pending ()) return

      ! transfer settings from shower_settings to shower
      call shower_set_D_Min_t (shower_settings%ps_mass_cutoff**2)
      call shower_set_D_Lambda_fsr (shower_settings%ps_fsr_lambda)
      call shower_set_D_Lambda_isr (shower_settings%ps_isr_lambda)
      call shower_set_D_Nf (shower_settings%ps_max_n_flavors)
      call shower_set_D_running_alpha_s_fsr &
           (shower_settings%ps_fsr_alpha_s_running)
      call shower_set_D_running_alpha_s_isr &
           (shower_settings%ps_isr_alpha_s_running)
      call shower_set_D_constantalpha_s &
           (shower_settings%ps_fixed_alpha_s)
      call shower_set_isr_pt_ordered &
           (shower_settings%ps_isr_pt_ordered)
      call shower_set_isr_angular_ordered &
           (shower_settings%ps_isr_angular_ordered)
      Call shower_set_primordial_kt_width &
           (shower_settings%ps_isr_primordial_kt_width)
      call shower_set_primordial_kt_cutoff &
           (shower_settings%ps_isr_primordial_kt_cutoff)
      call shower_set_maxz_isr (shower_settings%ps_isr_z_cutoff)
      call shower_set_minenergy_timelike (shower_settings%ps_isr_minenergy)
      call shower_set_tscalefactor_isr (shower_settings%ps_isr_tscalefactor)
      call shower_set_isr_only_onshell_emitted_partons &
           (shower_settings%ps_isr_only_onshell_emitted_partons)
      call shower_set_pdf_set (pdf_set)
      call shower_set_pdf_func (pdf_func)

      if (.not.msg_written) then
         call msg_message ("Using WHIZARD's internal showering")
         msg_written = .true.
      end if

      n_loop = 0
      TRY_SHOWER: do ! just a loop to be able to discard events
         n_loop = n_loop + 1
         if (n_loop > 1000) call msg_fatal &
              ("Shower: too many loops (try_shower)")
         call shower%create ()
         if (signal_is_pending ()) return
         max_color_nr = 0

         n_hadrons = 0
         n_in = 0
         n_out = 0
         do i = 1, particle_set_get_n_tot (particle_set)
```

2388

```
      if (particle_get_status (particle_set_get_particle &
           (particle_set, i)) == PRT_BEAM) &
           n_hadrons = n_hadrons + 1
      if (particle_get_status (particle_set_get_particle &
           (particle_set, i)) == PRT_INCOMING) &
           n_in = n_in + 1
      if (particle_get_status (particle_set_get_particle &
           (particle_set, i)) == PRT_OUTGOING) &
           n_out = n_out + 1
end do

allocate (connections (1:particle_set_get_n_tot (particle_set)))
connections = 0

allocate (hadrons (1:2))
allocate (partons (1:n_in+n_out))
allocate (parton_pointers (1:n_in+n_out))

j=0
if (n_hadrons > 0) then
   ! Transfer hadrons
   do i = 1, particle_set_get_n_tot (particle_set)
      if (particle_get_status (particle_set_get_particle &
           (particle_set, i)) == PRT_BEAM) then
         j = j+1
         hadrons(j)%nr = shower%get_next_free_nr ()
         hadrons(j)%momentum = particle_get_momentum &
              (particle_set_get_particle (particle_set, i))
         hadrons(j)%t = hadrons(j)%momentum**2
         hadrons(j)%type = particle_get_pdg &
              (particle_set_get_particle (particle_set, i))
         col_array=particle_get_color (particle_set_get_particle &
              (particle_set, i))
         hadrons(j)%c1 = col_array(1)
         hadrons(j)%c2 = col_array(2)
         max_color_nr = max (max_color_nr, abs(hadrons(j)%c1), &
              abs(hadrons(j)%c2))
         hadrons(j)%interactionnr = 1
         connections(i)=j
      end if
   end do
end if

! transfer incoming partons
j = 0
do i = 1, particle_set_get_n_tot (particle_set)
   if (particle_get_status (particle_set_get_particle &
        (particle_set, i)) == PRT_INCOMING) then
      j = j+1
      partons(j)%nr = shower%get_next_free_nr ()
      partons(j)%momentum = particle_get_momentum &
           (particle_set_get_particle (particle_set, i))
      partons(j)%t = partons(j)%momentum**2
      partons(j)%type = particle_get_pdg &
```

```
                (particle_set_get_particle (particle_set, i))
        col_array=particle_get_color &
            (particle_set_get_particle (particle_set, i))
        partons(j)%c1 = col_array (1)
        partons(j)%c2 = col_array (2)
        parton_pointers(j)%p => partons(j)
        max_color_nr = max (max_color_nr, abs (partons(j)%c1), &
            abs (partons(j)%c2))
        connections(i)=j
        ! insert dependences on hadrons
        if (particle_get_n_parents (particle_set_get_particle &
            (particle_set, i))==1) then
          parent = particle_get_parents (particle_set_get_particle &
              (particle_set, i))
          partons(j)%initial => hadrons (connections (parent(1)))
          partons(j)%x = space_part_norm (partons(j)%momentum) / &
                          space_part_norm (partons(j)%initial%momentum)
        end if
      end if
  end do
  if (signal_is_pending ()) return
  !!! transfer outgoing partons
  do i = 1, particle_set_get_n_tot (particle_set)
      if (particle_get_status (particle_set_get_particle &
          (particle_set, i)) == PRT_OUTGOING) then
        j = j + 1
        partons(j)%nr = shower%get_next_free_nr ()
        partons(j)%momentum = particle_get_momentum &
            (particle_set_get_particle (particle_set, i))
        partons(j)%t = partons(j)%momentum**2
        partons(j)%type = particle_get_pdg &
            (particle_set_get_particle (particle_set, i))
        col_array=particle_get_color &
            (particle_set_get_particle (particle_set, i))
        partons(j)%c1 = col_array(1)
        partons(j)%c2 = col_array(2)
        parton_pointers(j)%p => partons(j)
        max_color_nr = max (max_color_nr, abs &
            (partons(j)%c1), abs (partons(j)%c2))
        connections(i) = j
      end if
  end do

  deallocate (connections)

  ! insert these partons in shower
  call shower%set_next_color_nr (1 + max_color_nr)
  call shower%add_interaction_2ton_CKKW &
      (parton_pointers, shower_settings%ckkw_weights)

  if (signal_is_pending ()) return
  if (shower_settings%muli_active) then
    !!! Initialize muli pdf sets, unless initialized
    if (mi%is_initialized ()) then
```

```
            call mi%restart ()
      else
         if (debug) then
            ! call shower%write ()
            ! print *, "--------------"
            ! call interaction_write (shower%interactions(i)%i)
            ! print *, "--------------"
            ! call vector4_write &
            !    (shower%interactions(1)%i%partons(1)%p%momentum)
            ! call vector4_write &
            !    (shower%interactions(1)%i%partons(2)%p%momentum)
            ! print *, "--------------"
         end if
         call mi%initialize (&
               GeV2_scale_cutoff=D_Min_t, &
               GeV2_s=shower_interaction_get_s &
               (shower%interactions(1)%i), &
               muli_dir=char(os_data%whizard_mulipath))
      end if

      !!! initial interaction
      call mi%apply_initial_interaction ( &
            GeV2_s=shower_interaction_get_s(shower%interactions(1)%i), &
            x1=shower%interactions(1)%i%partons(1)%p%parent%x, &
            x2=shower%interactions(1)%i%partons(2)%p%parent%x, &
            pdg_f1=shower%interactions(1)%i%partons(1)%p%parent%type, &
            pdg_f2=shower%interactions(1)%i%partons(2)%p%parent%type, &
            n1=shower%interactions(1)%i%partons(1)%p%parent%nr, &
            n2=shower%interactions(1)%i%partons(2)%p%parent%nr)
   end if

   if (signal_is_pending ()) return

   if (shower_settings%ckkw_matching) then
      ! CKKW Matching
      call ckkw_matching_apply (shower, &
            shower_settings%ckkw_settings, &
            shower_settings%ckkw_weights, vetoed)
      if (vetoed) then
         return
      end if
   end if

   if (shower_settings%ps_isr_active) then
      i = 0
      BRANCHINGS: do
         i = i+1
         if (signal_is_pending ()) return
         if (shower_settings%muli_active) then
            call mi%generate_gev2_pt2 &
                  (shower%get_ISR_scale (), mi_scale)
         else
            mi_scale = 0.0
            end if
```

```
!!! Shower: debugging
!!! shower%generate_next_isr_branching returns a pointer to
!!! the parton with the next ISR-branching, this parton's
!!! scale is the scale of the next branching
! temppp=shower%generate_next_isr_branching_veto ()
temppp = shower%generate_next_isr_branching ()

if (.not. associated (temppp%p) .and. &
    mi_scale < D_Min_t) then
   exit BRANCHINGS
end if
!!! check if branching or interaction occurs next
if (associated (temppp%p)) then
   ps_scale = abs(temppp%p%t)
else
   ps_scale = 0._default
end if
if (mi_scale > ps_scale) then
   !!! discard branching evolution lower than mi_scale
   call shower%set_max_ISR_scale (mi_scale)
   if (associated (temppp%p)) &
        call parton_set_simulated(temppp%p, .false.)

   !!! execute new interaction
   deallocate (partons)
   deallocate (parton_pointers)
   allocate (partons(1:4))
   allocate (parton_pointers(1:4))
   do j = 1, 4
      partons(j)%nr = shower%get_next_free_nr ()
      partons(j)%belongstointeraction = .true.
      parton_pointers(j)%p => partons(j)
   end do
   call mi%generate_partons (partons(1)%nr, partons(2)%nr, &
        partons(1)%x, partons(2)%x, &
        partons(1)%type, partons(2)%type, &
        partons(3)%type, partons(4)%type)
   !!! calculate momenta
   shat = partons(1)%x *partons(2)%x * &
        shower_interaction_get_s(shower%interactions(1)%i)
   call parton_set_momentum (partons(1), &
        0.5_default*sqrt(shat), 0._default, 0._default, &
        0.5_default*sqrt(shat))
   call parton_set_momentum (partons(2), &
        0.5_default*sqrt(shat), 0._default, 0._default, &
        -0.5_default*sqrt(shat))
   call parton_set_initial (partons(1), &
        shower%interactions(1)%i%partons(1)%p%initial)
   call parton_set_initial (partons(2), &
        shower%interactions(1)%i%partons(2)%p%initial)
   partons(1)%belongstoFSR = .false.
   partons(2)%belongstoFSR = .false.
   !!! calculate color connections
```

```
        call mi%get_color_correlations &
             (shower%get_next_color_nr (), &
             max_color_nr,color_corr)
        call shower%set_next_color_nr (max_color_nr)

        partons(1)%c1 = color_corr(1,1)
        partons(1)%c2 = color_corr(2,1)
        partons(2)%c1 = color_corr(1,2)
        partons(2)%c2 = color_corr(2,2)
        partons(3)%c1 = color_corr(1,3)
        partons(3)%c2 = color_corr(2,3)
        partons(4)%c1 = color_corr(1,4)
        partons(4)%c2 = color_corr(2,4)

        call tao_random_number (phi)
        phi = 2*pi*phi
        call parton_set_momentum (partons(3), &
             0.5_default*sqrt(shat), sqrt(mi_scale)*cos(phi), &
             sqrt(mi_scale)*sin(phi), sqrt(0.25_default*shat - &
             mi_scale))
        call parton_set_momentum (partons(4), &
             0.5_default*sqrt(shat), -sqrt(mi_scale)*cos(phi), &
             -sqrt(mi_scale)*sin(phi), -sqrt(0.25_default*shat - &
             mi_scale))
        partons(3)%belongstoFSR = .true.
        partons(4)%belongstoFSR = .true.

        call shower%add_interaction_2ton (parton_pointers)
        n_int = size (shower%interactions)
        do k = 1, 2
           call mi%replace_parton &
             (shower%interactions(n_int)%i%partons(k)%p%initial%nr, &
              shower%interactions(n_int)%i%partons(k)%p%nr, &
              shower%interactions(n_int)%i%partons(k)%p%parent%nr, &
              shower%interactions(n_int)%i%partons(k)%p%type, &
              shower%interactions(n_int)%i%partons(k)%p%x, &
              mi_scale)
        end do
        call shower%write ()
     else
        !!! execute the next branching 'found' in the previous step
        call shower%execute_next_isr_branching (temppp)
        if (shower_settings%muli_active) then
           call mi%replace_parton (temppp%p%initial%nr, &
                temppp%p%child1%nr, temppp%p%nr, &
                temppp%p%type, temppp%p%x, ps_scale)
        end if

        if (debug)  call shower%write ()
     end if
end do BRANCHINGS

call shower%generate_fsr_for_isr_partons ()
if (debug)  call shower%write ()
```

```
      else
         if (signal_is_pending ()) return
         call shower%simulate_no_isr_shower ()
      end if

      !!! some bookkeeping, needed after the shower is done
      call shower%boost_to_labframe ()
      call shower%generate_primordial_kt ()
      call shower%update_beamremnants ()
      !!! clean-up muli: we should finalize the muli pdf sets when
      !!!       all runs are done.
      ! call mi%finalize ()

      if (shower_settings%ps_fsr_active) then
         do i = 1, size (shower%interactions)
            if (signal_is_pending ()) return
            call shower%interaction_generate_fsr_2ton &
                 (shower%interactions(i)%i)
         end do
      else
         call shower%simulate_no_fsr_shower ()
      end if
      if (debug) then
         call shower%write ()
         write (*, "(A)")  "SHOWER_FINISHED"
      end if

      if (shower_settings%mlm_matching) then
         !!! transfer momenta of the partons in the final state of
         !!!         the hard initeraction
         if (signal_is_pending ()) return
         if (allocated (JETS_ME))  deallocate (JETS_ME)
         call shower%get_final_colored_ME_partons (final_ME_partons)
         if (allocated (final_ME_partons)) then
            allocate (JETS_ME(1:size (final_ME_partons)))
            do i = 1, size (final_ME_partons)
               !!! transfer
               JETS_ME(i) = final_ME_partons(i)%p%momentum
            end do
            deallocate (final_ME_partons)
         end if
      end if

      u_S2W = free_unit ()
      if (debug .and. to_file) then
         open (unit=u_S2W, file="showerout.lhe", &
              status="replace", action="readwrite")
      else
         open (unit=u_S2W, status="scratch", action="readwrite")
      end if
      call shower%write_lhef (u_S2W)
      call shower_add_lhef_to_particle_set &
           (particle_set, u_S2W, model_in, os_data)
      close (u_S2W)
```

2394

```
      !!! move the particle data to the PYTHIA COMMON BLOCKS in case
      !!! hadronization is active
      if (shower_settings%hadronization_active) then
         if (signal_is_pending ()) return
         call shower_converttopythia (shower)
      end if
      deallocate (partons)
      deallocate (parton_pointers)
      exit TRY_SHOWER
   end do TRY_SHOWER
   if (debug) then
      call particle_set_write (particle_set)
      print *, &
        "--------------------apply_shower_particle_set------------------"
      print *, &
        "--------------------------------------------------------------"
      if (size (shower%interactions) >= 2) then
         call shower%write ()
      end if
   end if

   call shower%final ()
   !!! clean-up muli: we should finalize the muli pdf sets
   !!!      when all runs are done.
   ! call mi%finalize()
   return
end subroutine apply_WHIZARDshower_particle_set

subroutine get_ME_momenta_from_PYTHIA (JETS_ME)
  IMPLICIT DOUBLE PRECISION(A-H, O-Z)
  IMPLICIT INTEGER(I-N)
  COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
  SAVE /PYJETS/

  type(vector4_t), dimension(:), allocatable :: JETS_ME
  real(kind=default), dimension(:,:), allocatable :: pdum
  integer :: i, j, n_jets

  if (allocated (JETS_ME))  deallocate (JETS_ME)
  if (allocated (pdum))  deallocate (pdum)

  if (signal_is_pending ()) return
  !!! final ME partons start in 7th row of event record
  i = 7
  !!! find number of jets
  n_jets = 0
  do
     if (K(I,1) /= 21) exit
     if ((K(I,2) == 21) .or. (abs(K(I,2)) <= 6)) then
        n_jets = n_jets + 1
     end if
     i = i + 1
  end do
```

```
   if (n_jets == 0) return
   allocate (JETS_ME(1:n_jets))
   allocate (pdum(1:n_jets,4))

   !!! transfer jets
   i = 7
   j = 1
   pdum = p
   do
      if (K(I,1) /= 21) exit
      if ((K(I,2) == 21) .or. (abs(K(I,2)).le.6)) then
         JETS_ME(j)= vector4_moving (pdum(I,4), &
           vector3_moving ( [pdum(I,1),pdum(I,2),pdum(I,3)] ))
         j = j + 1
      end if
      i = i + 1
   end do
end subroutine get_ME_momenta_from_PYTHIA

subroutine matching_transfer_PS &
     (data, particle_set, settings)
  !!! transfer partons after parton shower to data%P_PS
  type(mlm_matching_data_t), intent(inout) :: data
  type(particle_set_t), intent(in) :: particle_set
  type(shower_settings_t), intent(in) :: settings
  integer :: i, j, n_jets_PS
  integer, dimension(2) :: col
  type(particle_t) :: tempprt
  real(double) :: eta, E, pl
  type(vector4_t) :: p_tmp

  !!! loop over particles and extract final colored ones with eta<etamax
  n_jets_PS = 0
  do i = 1, particle_set_get_n_tot (particle_set)
     if (signal_is_pending ()) return
     tempprt = particle_set_get_particle (particle_set, i)
     if (particle_get_status (tempprt) /= PRT_OUTGOING) cycle
     col = particle_get_color (tempprt)
     if (all (col == 0)) cycle
     if (data%is_hadron_collision) then
        p_tmp = particle_get_momentum (tempprt)
        if (energy (p_tmp) - longitudinal_part (p_tmp) < 1.E-10_default .or. &
            energy (p_tmp) + longitudinal_part (p_tmp) < 1.E-10_default) then
           eta = pseudorapidity (p_tmp)
        else
           eta = rapidity (p_tmp)
        end if
        if (eta > settings%ms%mlm_etaClusfactor * &
             settings%ms%mlm_etamax)  then
           if (debug) then
              print *, "REJECTING"
              call particle_write (tempprt)
           end if
```

```fortran
            cycle
         end if
      end if
      n_jets_PS = n_jets_PS + 1
   end do

   allocate (data%P_PS(1:n_jets_PS))
   if (debug)  write (*, "(A,1x,I0)")  "n_jets_ps =", n_jets_ps

   j = 1
   do i = 1, particle_set_get_n_tot (particle_set)
      tempprt = particle_set_get_particle (particle_set, i)
      if (particle_get_status (tempprt) /= PRT_OUTGOING) cycle
      col = particle_get_color (tempprt)
      if(all(col == 0)) cycle
      if (data%is_hadron_collision) then
         p_tmp = particle_get_momentum (tempprt)
         if (energy (p_tmp) - longitudinal_part (p_tmp) < 1.E-10_default .or. &
              energy (p_tmp) + longitudinal_part (p_tmp) < 1.E-10_default) then
            eta = pseudorapidity (p_tmp)
         else
            eta = rapidity (p_tmp)
         end if
         if (eta > settings%ms%mlm_etaClusfactor * &
              settings%ms%mlm_etamax) cycle
      end if
      data%P_PS(j) = particle_get_momentum (tempprt)
      j = j + 1
   end do
end subroutine matching_transfer_PS

subroutine apply_PYTHIAhadronization &
     (particle_set, shower_settings, model, valid)
  type(particle_set_t), intent(inout) :: particle_set
  type(shower_settings_t), intent(in) :: shower_settings
  type(model_t), pointer, intent(in) :: model
  logical, intent(inout) :: valid
  integer :: u_W2P, u_P2W
  type(string_t) :: remaining_PYGIVE, partial_PYGIVE
  logical, save :: msg_written = .false.
  character*10 buffer

  if (.not. shower_settings%hadronization_active)  return
  if (.not. valid) return
  if (signal_is_pending ()) return

  u_W2P = free_unit ()
  if (debug) then
     open (unit=u_W2P, status="replace", file="whizardout.lhe", &
           action="readwrite")
  else
     open (unit=u_W2P, status="scratch", action="readwrite")
  end if
  call shower_W2P_write_event (u_W2P)
```

```
rewind (u_W2P)
write (buffer, "(I10)")  u_W2P
call PYGIVE ("MSTP(161)=" // buffer)
call PYGIVE ("MSTP(162)=" // buffer)

!!! Assume that the event is still present in the PYTHIA common blocks
! call pygive ("MSTP(61)=0")  ! switch off ISR
! call pygive ("MSTP(71)=0")  ! switch off FSR

if (.not. shower_settings%ps_use_PYTHIA_shower .and. &
     len(shower_settings%ps_PYTHIA_PYGIVE) > 0) then
   remaining_PYGIVE = shower_settings%ps_PYTHIA_PYGIVE
   do while (len(remaining_PYGIVE) > 0)
      if (signal_is_pending ()) return
      call split (remaining_PYGIVE, partial_PYGIVE, ";")
      call PYGIVE (char (partial_PYGIVE))
   end do
   if (shower_get_PYTHIA_error () /= 0) then
      call msg_fatal ("PYTHIA didn't recognize ps_PYTHIA_PYGIVE setting")
   end if
end if

if (.not. (shower_settings%ps_use_PYTHIA_shower .and. &
     (shower_settings%ps_isr_active.or. &
     shower_settings%ps_fsr_active))) then
   if (len(shower_settings%ps_PYTHIA_PYGIVE) > 0) then
      remaining_PYGIVE = shower_settings%ps_PYTHIA_PYGIVE
      do while (len(remaining_PYGIVE) > 0)
         if (signal_is_pending ()) return
         call split (remaining_PYGIVE, partial_PYGIVE, ";")
         call PYGIVE (char(partial_PYGIVE))
      end do
      if (shower_get_PYTHIA_error () /= 0) then
         call msg_fatal &
              ("PYTHIA did not recognize ps_PYTHIA_PYGIVE setting")
      end if
   end if
end if

if (.not.msg_written) then
   call msg_message &
        ("Using PYTHIA interface for hadronization and decays")
   msg_written = .true.
end if

call PYGIVE ("MSTP(111)=1") !!! switch on hadronization
if (signal_is_pending ()) return
call PYEXEC

if (shower_get_PYTHIA_error () > 0) then
   !!! clean up, discard shower and exit
   call shower_set_PYTHIA_error (0)
   close (u_W2P)
   valid = .false.
```

2398

```
      else
         !!! convert back
         u_P2W = free_unit ()
         write (buffer, "(I10)")  u_P2W
         call PYGIVE ("MSTP(163)=" // buffer)
         if (debug .and. to_file) then
            open (unit=u_P2W, file="pythiaout2.lhe", status="replace", &
                  action="readwrite")
         else
            open (unit=u_P2W, status="scratch", action="readwrite")
         end if
         !!! convert pythia /PYJETS/ to lhef given in MSTU(163)=u1
         call pylheo
         !!! read and add lhef from u_P2W
         if (signal_is_pending ()) return
         call shower_add_lhef_to_particle_set &
               (particle_set, u_P2W, model, os_data)
         close (u_W2P)
         close (u_P2W)
         valid = .true.
      end if
   end subroutine apply_PYTHIAhadronization

   subroutine shower_W2P_write_event (unit)
      integer, intent(in) :: unit
      type(xml_tag_t), allocatable :: tag_lhef, tag_head, tag_init, &
            tag_event, tag_gen_n, tag_gen_v
      allocate (tag_lhef, tag_head, tag_init, tag_event, &
            tag_gen_n, tag_gen_v)
      call tag_lhef%init (var_str ("LesHouchesEvents"), &
         [xml_attribute (var_str ("version"), var_str ("1.0"))], .true.)
      call tag_head%init (var_str ("header"), .true.)
      call tag_init%init (var_str ("init"), .true.)
      call tag_event%init (var_str ("event"), .true.)
      call tag_gen_n%init (var_str ("generator_name"), .true.)
      call tag_gen_v%init (var_str ("generator_version"), .true.)
      call tag_lhef%write (unit); write (unit, *)
      call tag_head%write (unit); write (unit, *)
      write (unit, "(2x)", advance = "no")
      call tag_gen_n%write (var_str ("WHIZARD"), unit)
      write (unit, *)
      write (unit, "(2x)", advance = "no")
      call tag_gen_v%write (var_str ("⟨Version⟩"), unit)
      write (unit, *)
      call tag_head%close (unit); write (unit, *)
      call tag_init%write (unit); write (unit, *)
      call heprup_write_lhef (unit)
      call tag_init%close (unit); write (unit, *)
      call tag_event%write (unit); write (unit, *)
      call hepeup_write_lhef (unit)
      call tag_event%close (unit); write (unit, *)
      call tag_lhef%close (unit); write (unit, *)
      deallocate (tag_lhef, tag_head, tag_init, tag_event, &
            tag_gen_n, tag_gen_v)
```

```
      end subroutine shower_W2P_write_event

    end subroutine apply_shower_particle_set
```

Read in the LHE file opened in unit `u` and add the final particles to the `particle_set`, the outgoing particles of the existing `particle_set` are set to be virtual before the outgoing particles from the LHEF are added to the `particle_set` as outgoing particles. At the same time, it is searched for pairs of existing and new particles with equal momenta and PDG code, these pairs are marked as connected. All remaining existing former outgoing particles are set to be the mothers of all remaining new outgoing particles.

⟨*Shower: procedures*⟩+≡
```
    subroutine shower_add_lhef_to_particle_set &
        (particle_set, u, model_in, os_data)
      type(particle_set_t), intent(inout) :: particle_set
      integer, intent(in) :: u
      type(model_t), intent(in), pointer :: model_in
      type(model_t), target, save :: model_SM_hadrons
      type(model_t), pointer :: model
      logical, save :: model_SM_hadrons_associated = .false.
      type(os_data_t), intent(in) :: os_data
      logical :: exist_SM_hadrons
      type(string_t) :: filename
      type(flavor_t) :: flv
      type(color_t) :: col
      logical :: logging_save

      integer :: newsize, oldsize
      type(particle_t), dimension(:), allocatable :: temp_prt
      integer :: i, j
      integer :: n_available_parents;
      integer, dimension(:), allocatable :: available_parents
      integer, dimension(:), allocatable :: available_children
      logical, dimension(:), allocatable :: direct_child
      type(vector4_t) :: diffmomentum
      integer, PARAMETER :: MAXLEN=200
      CHARACTER*(MAXLEN) STRING
      integer ibeg
      INTEGER :: NUP,IDPRUP,IDUP,ISTUP
      real(kind=double) :: XWGTUP,SCALUP,AQEDUP,AQCDUP,VTIMUP,SPINUP
      integer :: MOTHUP(1:2),ICOLUP(1:2)
      real(kind=double) :: PUP(1:5)
      real(kind=default) :: pup_dum(1:5)
      character*5 buffer

      CHARACTER*6 STRFMT
      STRFMT='(A000)'
      WRITE(STRFMT(3:5),'(I3)') MAXLEN

      rewind (u)

      !!! get newsize of particle_set, newsize = old size of
      !!!    particle_set + #entries - 2 (incoming partons in lhef)
      oldsize = particle_set_get_n_tot (particle_set)
```

```
!!! Loop until finds line beginning with "<event>" or "<event ".
do
   read (u,*,END=501,ERR=502) STRING
   IBEG=0
   do
      if (signal_is_pending ()) return
      IBEG = IBEG + 1
      ! Allow indentation.
      IF (STRING(IBEG:IBEG).EQ.' ' .and. IBEG < MAXLEN-6) cycle
      exit
   end do
   IF (string(IBEG:IBEG+6) /= '<event>' .and. &
         string(IBEG:IBEG+6) /= '<event ') cycle
   exit
end do
!!! Read first line of event info -> number of entries
read (u, *, END=503, ERR=504) NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
newsize = oldsize + NUP - 2
allocate (temp_prt (1:newsize))

allocate (available_parents (1:oldsize))
available_parents = 0
do i = 1, particle_set_get_n_tot (particle_set)
   if (signal_is_pending ()) return
   temp_prt (i) = particle_set_get_particle (particle_set, i)
   if (particle_get_status (temp_prt (i)) == PRT_OUTGOING .or. &
         particle_get_status (temp_prt (i)) == PRT_BEAM_REMNANT) then
      call particle_reset_status (temp_prt (i), PRT_VIRTUAL)
      available_parents (i) = i
   end if
end do

allocate (available_children (1:newsize))
allocate (direct_child (1:newsize))
available_children = 0
direct_child = .false.

!!! transfer particles from lhef to particle_set
!!!...Read NUP subsequent lines with information on each particle.
DO I = 1, NUP
   READ (u,*,END=200,ERR=505) IDUP, ISTUP, MOTHUP(1), MOTHUP(2), &
         ICOLUP(1), ICOLUP(2), (PUP (J),J=1,5), VTIMUP, SPINUP
   if ((I.eq.1).or.(I.eq.2)) cycle

   call particle_reset_status (temp_prt(oldsize+i-2), PRT_OUTGOING)
   !!! Settings for unpolarized particles
   ! particle_set%prt (oldsize+i-2)%polarization = 0 ! =PRT_UNPOLARIZED !??
   if (model_test_particle (model_in, IDUP)) then
      model => model_in
   else
      ! prepare model_SM_hadrons for hadrons created in the hadronization
      ! and not present in the model file
      if (.not. model_SM_hadrons_associated) then
         ! call os_data_init (os_data)
```

```
         filename = "SM_hadrons.mdl"
         logging_save = logging
         logging = .false.
         call model_read (model_SM_hadrons, filename, os_data, &
              exist_SM_hadrons)
         logging = logging_save
         model_SM_hadrons_associated = .true.
      end if
      if (model_test_particle (model_SM_hadrons, IDUP)) then
         model => model_SM_hadrons
      else
         write (buffer, "(I5)") IDUP
         call msg_error ("Parton " // buffer // &
              " found neither in given model file nor in SM_hadrons")
         return
      end if
   end if
   call flavor_init (flv, IDUP, model)
   call particle_set_flavor (temp_prt (oldsize+i-2), flv)

   if (IABS(IDUP) == 2212 .or. IABS(IDUP) == 2112) then
      ! PYTHIA sometimes sets color indices for protons and neutrons (?)
      ICOLUP (1) = 0
      ICOLUP (2) = 0
   end if
   call color_init_col_acl (col, ICOLUP (1), ICOLUP (2))
   call particle_set_color (temp_prt (oldsize+i-2), col)
   !!! Settings for unpolarized particles
   ! particle_set%prt (oldsize+i-2)%hel = ??
   ! particle_set%prt (oldsize+i-2)%pol = ??
   pup_dum = PUP
   call particle_set_momentum (temp_prt (oldsize+i-2), &
        vector4_moving (pup_dum (4), &
        vector3_moving ([pup_dum (1), pup_dum (2), pup_dum (3)])))

   available_children (oldsize+i-2) = oldsize+i-2
   !!! search for an existing particle with the same momentum
   !!!   -> treat these as mother and daughter
   do j = 1, size (available_parents)
      if (available_parents (j) == 0) cycle
      diffmomentum = particle_get_momentum &
           (temp_prt (available_parents (j))) - &
           particle_get_momentum (temp_prt(oldsize+i-2))
      if (abs(diffmomentum**2) < 1E-10_default .and. &
           particle_get_pdg (temp_prt (available_parents (j))).eq. &
           particle_get_pdg (temp_prt (oldsize+i-2))) then
         direct_child (available_parents (j)) = .true.
         direct_child (oldsize+i-2) = .true.
         call particle_set_parents (temp_prt (oldsize+i-2), &
              [available_parents(j)] )
         call particle_set_children (temp_prt (available_parents(j)), &
              [oldsize+i-2] )
         available_parents (j) = 0
         available_children (oldsize+i-2) = 0
```

2402

```
            end if
          end do
        end do

        !!! remove zeros in available parents and available children
        available_parents  = pack (available_parents , available_parents  /= 0)
        available_children = pack (available_children, available_children /= 0)

        do i = 1, size (available_parents)
          if (direct_child (available_parents (i))) cycle
          call particle_set_children &
               (temp_prt (available_parents (i)), available_children)
        end do
        do i = oldsize + 1, newsize
           if (direct_child (i)) cycle
           call particle_set_parents (temp_prt(i), available_parents)
        end do

        ! transfer to particle_set
        call particle_set_replace (particle_set, temp_prt)
        if (allocated (available_children)) deallocate (available_children)
        if (allocated (available_parents))  deallocate (available_parents)
        deallocate (direct_child)
        deallocate (temp_prt)
        call model_final (model_SM_hadrons)

  200 continue
      return

  501 write(*,*) "READING LHEF failed 501"
      return
  502 write(*,*) "READING LHEF failed 502"
      return
  503 write(*,*) "READING LHEF failed 503"
      return
  504 write(*,*) "READING LHEF failed 504"
      return
  505 write(*,*) "READING LHEF failed 504"
      return
    end subroutine shower_add_lhef_to_particle_set
```
⟨*Shower: procedures*⟩+≡
```
  !!!!!!!!!!!PYTHIA STYLE!!!!!!!!!!!!!!!
  !!! originally PYLHEF subroutine from PYTHIA 6.4.22

  !C...Write out the showered event to a Les Houches Event File.
  !C...Take MSTP(161) as the input for <init>...</init>

        SUBROUTINE PYLHEO

  !C...Double precision and integer declarations.
        IMPLICIT DOUBLE PRECISION(A-H, O-Z)
        IMPLICIT INTEGER(I-N)

  !C...PYTHIA commonblock: only used to provide read/write units and version.
```

```
      COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
      COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
      SAVE /PYPARS/
      SAVE /PYJETS/

!C...User process initialization commonblock.
      INTEGER MAXPUP
      PARAMETER (MAXPUP=100)
      INTEGER IDBMUP,PDFGUP,PDFSUP,IDWTUP,NPRUP,LPRUP
      DOUBLE PRECISION EBMUP,XSECUP,XERRUP,XMAXUP
      COMMON/HEPRUP/IDBMUP(2),EBMUP(2),PDFGUP(2),PDFSUP(2),IDWTUP,NPRUP,XSECUP(MAXPUP),XERRUP(MAXP
      SAVE /HEPRUP/

!C...User process event common block.
      INTEGER MAXNUP
      PARAMETER (MAXNUP=500)
      INTEGER NUP,IDPRUP,IDUP,ISTUP,MOTHUP,ICOLUP
      PARAMETER (KSUSY1=1000000,KSUSY2=2000000,KTECHN=3000000, &
          KEXCIT=4000000,KDIMEN=5000000)
      DOUBLE PRECISION XWGTUP,SCALUP,AQEDUP,AQCDUP,PUP,VTIMUP,SPINUP
      COMMON/HEPEUP/NUP,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP,IDUP(MAXNUP),ISTUP(MAXNUP),MOTHUP(2,MAX
                   PUP(5,MAXNUP),VTIMUP(MAXNUP),SPINUP(MAXNUP)
      SAVE /HEPEUP/

!C...Lines to read in assumed never longer than 200 characters.
      PARAMETER (MAXLEN=200)
      CHARACTER*(MAXLEN) STRING

      INTEGER LEN

!C...Format for reading lines.
      CHARACTER*6 STRFMT
      STRFMT='(A000)'
      WRITE(STRFMT(3:5),'(I3)') MAXLEN

!C...Rewind initialization and event files.
      REWIND MSTP(161)
      REWIND MSTP(162)

!C...Write header info.
      WRITE(MSTP(163),'(A)') '<LesHouchesEvents version="1.0">'
      WRITE(MSTP(163),'(A)') '<!--'
      WRITE(MSTP(163),'(A,I1,A1,I3)') 'File generated with PYTHIA ',MSTP(181),'.',MSTP(182)
      WRITE(MSTP(163),'(A)') ' and the WHIZARD2 interface'
      WRITE(MSTP(163),'(A)') '-->'

!C...Loop until finds line beginning with "<init>" or "<init ".
  100 READ(MSTP(161),STRFMT,END=400,ERR=400) STRING
      IBEG=0
  110 IBEG=IBEG+1
!C...Allow indentation.
      IF(STRING(IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-5) GOTO 110
      IF(STRING(IBEG:IBEG+5).NE.'<init>'.AND.STRING(IBEG:IBEG+5).NE.'<init ') GOTO 100
```

```fortran
!C...Read first line of initialization info and get number of processes.
      READ(MSTP(161),'(A)',END=400,ERR=400) STRING
      READ(STRING,*,ERR=400) IDBMUP(1),IDBMUP(2),EBMUP(1),EBMUP(2),PDFGUP(1),PDFGUP(2),PDFSUP(1),P

!C...Copy initialization lines, omitting trailing blanks.
!C...Embed in <init> ... </init> block.
      WRITE(MSTP(163),'(A)') '<init>'
      DO IPR=0,NPRUP
        IF(IPR.GT.0) READ(MSTP(161),'(A)',END=400,ERR=400) STRING
        LEN=MAXLEN+1
  120   LEN=LEN-1
        IF(LEN.GT.1.AND.STRING(LEN:LEN).EQ.' ') GOTO 120
        WRITE(MSTP(163),'(A)',ERR=400) STRING(1:LEN)
      end DO
      WRITE(MSTP(163),'(A)') '</init>'

!!!! Find the numbers of entries of the <event block>
      NENTRIES = 2       ! incoming partons (nearest to the beam particles)
      DO I=1,N
         if((K(I,1).eq.1) .or. (K(I,1).eq.2)) then
            if(P(I,4) < 1D-10) cycle
            NENTRIES = NENTRIES + 1
         end if
      end DO

!C...Begin an <event> block. Copy event lines, omitting trailing blanks.
      WRITE(MSTP(163),'(A)') '<event>'
      WRITE(MSTP(163),*) NENTRIES,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP

      DO I=3,4        ! the incoming partons nearest to the beam particles
         WRITE(MSTP(163),*)  K(I,2),-1,0,0,0,0,(P(I,J),J=1,5),0, -9
      end DO
      NDANGLING_COLOR = 0
      NCOLOR = 0
      NDANGLING_ANTIC = 0
      NANTIC = 0
      NNEXTC = 1    ! TODO find next free color number ??
      DO I=1,N
         if (signal_is_pending ()) return
         if((K(I,1).eq.1) .or. (K(I,1).eq.2)) then
            ! workaround for zero energy photon in electron ISR
            if (P(I,4) < 1E-10_default) cycle
            if ((K(I,2).eq.21) .or. (IABS(K(I,2)) <= 8) .or. &
                (IABS(K(I,2)) >= KSUSY1+1 .and. IABS(K(I,2)) <= KSUSY1+8) &
                .or. &
                (IABS(K(I,2)) >= KSUSY2+1 .and. IABS(K(I,2)) <= KSUSY2+8) .or. &
                 (IABS(K(I,2)) >= 1000 .and. IABS(K(I,2)) <= 9999) ) then
               if(NDANGLING_COLOR.eq.0 .and. NDANGLING_ANTIC.eq.0) then
                  ! new color string
                  ! Gluon and gluino only color octets implemented so far
                  if(K(I,2).eq.21 .or. K(I,2).eq.1000021) then
                     NCOLOR = NNEXTC
                     NDANGLING_COLOR = NCOLOR
                     NNEXTC = NNEXTC + 1
```

```
                       NANTIC = NNEXTC
                       NDANGLING_ANTIC = NANTIC
                       NNEXTC = NNEXTC + 1
                   elseif(K(I,2) .gt. 0) then  ! particles to have color
                       NCOLOR = NNEXTC
                       NDANGLING_COLOR = NCOLOR
                       NANTIC = 0
                       NNEXTC = NNEXTC + 1
                   elseif(K(I,2) .lt. 0) then  ! antiparticles to have anticolor
                       NANTIC = NNEXTC
                       NDANGLING_ANTIC = NANTIC
                       NCOLOR = 0
                       NNEXTC = NNEXTC + 1
                   end if
               else if(K(I,1).eq.1) then
                   ! end of string
                   NCOLOR = NDANGLING_ANTIC
                   NANTIC = NDANGLING_COLOR
                   NDANGLING_COLOR = 0
                   NDANGLING_ANTIC = 0
               else
                   ! inside the string
                   if(NDANGLING_COLOR .ne. 0) then
                       NANTIC = NDANGLING_COLOR
                       NCOLOR = NNEXTC
                       NDANGLING_COLOR = NNEXTC
                       NNEXTC = NNEXTC +1
                   else if(NDANGLING_ANTIC .ne. 0) then
                       NCOLOR = NDANGLING_ANTIC
                       NANTIC = NNEXTC
                       NDANGLING_ANTIC = NNEXTC
                       NNEXTC = NNEXTC +1
                   else
                       print *, "ERROR IN PYLHEO"
                   end if
               end if
           else
               NCOLOR = 0
               NANTIC = 0
           end if

           !!! As no intermediate are given out here, assume the
           !!!   incoming partons to be the mothers
           WRITE(MSTP(163),*)  K(I,2),1,1,2,NCOLOR,NANTIC,(P(I,J),J=1,5),0, -9
       end if
   end DO

!C..End the <event> block. Loop back to look for next event.
      WRITE(MSTP(163),'(A)') '</event>'

!C...Successfully reached end of event loop: write closing tag
!C...and remove temporary intermediate files (unless asked not to).
320   WRITE(MSTP(163),'(A)') '</LesHouchesEvents>'
      RETURN
```

```
!!C...Error exit.
   400 WRITE(*,*) ' PYLHEO file joining failed!'

         RETURN
      END SUBROUTINE PYLHEO
```

17.3.4 Event Transform

⟨*Shower: public*⟩+≡
```
    public :: evt_shower_t
```

⟨*Shower: types*⟩+≡
```
    type, extends (evt_t) :: evt_shower_t
      type(shower_settings_t) :: settings
      type(os_data_t) :: os_data
      integer :: pdf_type = STRF_NONE
      integer :: pdf_set = 0
     contains
     ⟨Shower: evt shower: TBP⟩
    end type evt_shower_t
```

Output.

⟨*Shower: evt shower: TBP*⟩≡
```
    procedure :: write => evt_shower_write
```

⟨*Shower: procedures*⟩+≡
```
    subroutine evt_shower_write (object, unit, testflag)
      class(evt_shower_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      integer :: u
      u = output_unit (unit)
      call write_separator_double (u)
      write (u, "(1x,A)")  "Event transform: shower"
      call write_separator (u)
      call object%base_write (u, testflag = testflag)
      call write_separator (u)
      call object%settings%write (u)
    end subroutine evt_shower_write
```

Initialize the parameters.

⟨*Shower: evt shower: TBP*⟩+≡
```
    procedure :: init => evt_shower_init
```

⟨*Shower: procedures*⟩+≡
```
    subroutine evt_shower_init (evt, settings, os_data)
      class(evt_shower_t), intent(out) :: evt
      type(shower_settings_t), intent(in) :: settings
      type(os_data_t), intent(in) :: os_data
      evt%settings = settings
      evt%os_data = os_data
    end subroutine evt_shower_init
```

Use the process and beam structure to determine PDF parameters.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: setup_pdf => evt_shower_setup_pdf
```

⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_setup_pdf (evt, process, beam_structure)
  class(evt_shower_t), intent(inout) :: evt
  type(process_t), intent(in) :: process
  type(beam_structure_t), intent(in) :: beam_structure
  if (beam_structure%contains ("lhapdf")) then
     evt%pdf_type = STRF_LHAPDF
     evt%pdf_set = process%get_pdf_set ()
     write (msg_buffer, "(A,I0)")  "Shower: interfacing LHAPDF set #", &
          evt%pdf_set
     call msg_message ()
  else if (beam_structure%contains ("pdf_builtin")) then
     evt%pdf_type = STRF_PDF_BUILTIN
     evt%pdf_set = process%get_pdf_set ()
     write (msg_buffer, "(A,I0)")  "Shower: interfacing PDF builtin set #", &
          evt%pdf_set
     call msg_message ()
  end if
end subroutine evt_shower_setup_pdf
```

Prepare: nothing to do here.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: prepare_new_event => evt_shower_prepare_new_event
```

⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_prepare_new_event (evt, i_mci, i_term)
  class(evt_shower_t), intent(inout) :: evt
  integer, intent(in) :: i_mci, i_term
  call evt%reset ()
end subroutine evt_shower_prepare_new_event
```

Here we take the particle set from the previous event transform (assuming that there is always one) and apply the shower algorithm. The result is stored in the event transform of the current object. We always return a probability of unity as we don't have the analytic weight of the combination of shower, MLM matching and hadronization. A subdivision into multiple event transformations might be useful. Invalid or vetoed events have to be discarded by the caller which is why we mark the particle set as invalid.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: generate_weighted => evt_shower_generate_weighted
```

⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_generate_weighted (evt, probability)
  class(evt_shower_t), intent(inout) :: evt
  real(default), intent(out) :: probability
  logical :: valid, vetoed
  valid = .true.
  vetoed = .false.
  if (evt%previous%particle_set_exists) then
     evt%particle_set = evt%previous%particle_set
```

```
              if (evt%settings%ps_use_PYTHIA_shower .or. &
                 evt%settings%hadronization_active) then
                 call evt%assure_heprup ()
              end if
              if (evt%settings%ckkw_matching) then
                 call ckkw_pseudo_shower_weights_init (evt%settings%ckkw_weights)
                 call ckkw_fake_pseudo_shower_weights (evt%settings%ckkw_settings, &
                      evt%settings%ckkw_weights, evt%particle_set)
              end if
              call apply_shower_particle_set (evt%particle_set, &
                   evt%settings, evt%model, &
                   evt%os_data, evt%pdf_type, evt%pdf_set, valid, vetoed)
              probability = 1
              !!! BCN: WK please check: In 2.1.1 vetoed events reduced sim%n_events by
              ! one while invalid events did not. This bookkeeping should be reenabled
              ! to have the correct cross section / luminosity.
              if (valid .and. .not. vetoed) then
                 evt%particle_set_exists = .true.
              else
                 evt%particle_set_exists = .false.
              end if
           else
              call msg_bug ("Shower: input particle set does not exist")
           end if
        end subroutine evt_shower_generate_weighted
```

The particle set was already made by the routine above (or the event was vetoed). So we do nothing here. The factorization parameters are irrelevant.

⟨*Shower: evt shower: TBP*⟩+≡
```
    procedure :: make_particle_set => evt_shower_make_particle_set
```

⟨*Shower: procedures*⟩+≡
```
    subroutine evt_shower_make_particle_set &
        (evt, factorization_mode, keep_correlations, r)
      class(evt_shower_t), intent(inout) :: evt
      integer, intent(in) :: factorization_mode
      logical, intent(in) :: keep_correlations
      real(default), dimension(:), intent(in), optional :: r
    end subroutine evt_shower_make_particle_set
```

Matching and hadronization functions

The HEPRUP (event) common block is needed for the interface to the shower. Filling of it is triggered by some output file formats. If these are not present, the common block is filled with some dummy information. Be generous with the number of processes in HEPRUP so that PYTHIA only rarely needs to be reinitialized in case events with higher process ids are generated.

⟨*Shower: evt shower: TBP*⟩+≡
```
    procedure :: assure_heprup => event_shower_assure_heprup
```

⟨*Shower: procedures*⟩+≡
```
    subroutine event_shower_assure_heprup (evt)
```

```
      class(evt_shower_t), intent(in) :: evt
      integer :: i, num_id
      integer, parameter :: min_processes = 10
```

⟨*HEP common: parameters*⟩
⟨*HEP common: variables*⟩
⟨*HEP common: common blocks*⟩

```
    num_id = 1
    if (LPRUP (num_id) /= 0)  return

    call heprup_init ( &
         [ particle_get_pdg (particle_set_get_particle &
                               (evt%particle_set, 1)), &
           particle_get_pdg (particle_set_get_particle &
                               (evt%particle_set, 2)) ] , &
         [ vector4_get_component (particle_get_momentum &
             (particle_set_get_particle (evt%particle_set, 1)), 0), &
           vector4_get_component (particle_get_momentum &
             (particle_set_get_particle (evt%particle_set, 2)), 0) ], &
         num_id, .false., .false.)
      do i = 1, (num_id / min_processes + 1) * min_processes
         call heprup_set_process_parameters (i = i, process_id = &
              i, cross_section = 1._default, error = 1._default)
      end do
  end subroutine event_shower_assure_heprup
```

Generate fake ckkw weights. This can be dropped, once information from the matrix element generation is available.

⟨*Shower: public*⟩+≡
```
    public :: ckkw_fake_pseudo_shower_weights
```

⟨*Shower: procedures*⟩+≡
```
    subroutine ckkw_fake_pseudo_shower_weights &
         (ckkw_pseudo_shower_settings, &
          ckkw_pseudo_shower_weights, particle_set)
      type(ckkw_matching_settings_t), intent(inout) :: &
          ckkw_pseudo_shower_settings
      type(ckkw_pseudo_shower_weights_t), intent(inout) :: &
          ckkw_pseudo_shower_weights
      type(particle_set_t), intent(in) :: particle_set
      integer :: i, j, k
      integer :: n
      type(vector4_t) :: momentum

      ckkw_pseudo_shower_settings%alphaS = 1.0_default
      ckkw_pseudo_shower_settings%Qmin = 1.0_default
      ckkw_pseudo_shower_settings%n_max_jets = 3

      n = 2**particle_set_get_n_tot(particle_set)
      if (allocated (ckkw_pseudo_shower_weights%weights)) then
         deallocate (ckkw_pseudo_shower_weights%weights)
      end if
      allocate (ckkw_pseudo_shower_weights%weights (1:n))
```

```
      do i = 1, n
         momentum = vector4_null
         do j = 1, particle_set_get_n_tot (particle_set)
            if (btest (i,j-1)) then
               momentum = momentum + particle_get_momentum &
                    (particle_set_get_particle (particle_set, j))
            end if
         end do
         if (momentum**1 > 0.0) then
            ckkw_pseudo_shower_weights%weights(i) = 1.0 / (momentum**2)
         end if
      end do
      ! equally distribute the weights by type
      if (allocated (ckkw_pseudo_shower_weights%weights_by_type)) then
         deallocate (ckkw_pseudo_shower_weights%weights_by_type)
      end if
      allocate (ckkw_pseudo_shower_weights%weights_by_type (1:n, 0:4))
      do i = 1, n
         do j = 0, 4
            ckkw_pseudo_shower_weights%weights_by_type(i,j) = &
                 0.2 * ckkw_pseudo_shower_weights%weights(i)
         end do
      end do
    end subroutine ckkw_fake_pseudo_shower_weights
```

17.3.5 Unit Tests

⟨*Shower: public*⟩+≡
```
    public :: shower_test
```

⟨*Shower: tests*⟩≡
```
    subroutine shower_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
      ⟨*Shower: execute tests*⟩
```
    end subroutine shower_test
```

Testbed

This sequence sets up a two-jet process, ready for generating events.

⟨*Shower: tests*⟩+≡
```
    subroutine setup_testbed &
         (prefix, os_data, lib, model_list, model, process, process_instance)
      type(string_t), intent(in) :: prefix
      type(os_data_t), intent(out) :: os_data
      type(process_library_t), intent(out), target :: lib
      type(model_list_t), intent(out) :: model_list
      type(model_t), pointer, intent(out) :: model
      type(process_t), target, intent(out) :: process
      type(process_instance_t), target, intent(out) :: process_instance
      type(var_list_t), pointer :: model_vars
```

```
type(string_t) :: model_name, libname, procname, run_id
type(process_def_entry_t), pointer :: entry
type(string_t), dimension(:), allocatable :: prt_in, prt_out
type(qcd_t) :: qcd
class(rng_factory_t), allocatable :: rng_factory
class(prc_core_t), allocatable :: core_template
class(mci_t), allocatable :: mci_template
class(phs_config_t), allocatable :: phs_config_template
real(default) :: sqrts

model_name = "SM"
libname = prefix // "_lib"
procname = prefix // "p"
run_id = "1"

call os_data_init (os_data)
allocate (rng_tao_factory_t :: rng_factory)
call model_list%read_model (model_name, model_name // ".mdl", &
     os_data, model)
model_vars => model_get_var_list_ptr (model)
call var_list_set_real (model_vars, var_str ("me"), 0._default, &
     is_known = .true.)

call lib%init (libname)

allocate (prt_in (2), source = [var_str ("e-"), var_str ("e+")])
allocate (prt_out (2), source = [var_str ("d"), var_str ("dbar")])

allocate (entry)
call entry%init (procname, model, n_in = 2, n_components = 1)
call omega_make_process_component (entry, 1, &
     model_name, prt_in, prt_out, &
     report_progress=.true.)
call lib%append (entry)

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

call process%init (procname, run_id, lib, os_data, &
     qcd, rng_factory, model_list)

allocate (prc_omega_t :: core_template)
allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)

select type (core_template)
type is (prc_omega_t)
   call core_template%set_parameters (model = model)
end select
call process%init_component &
     (1, core_template, mci_template, phs_config_template)
```

```
      sqrts = 1000
      call process%setup_beams_sqrts (sqrts)
      call process%configure_phs ()
      call process%setup_mci ()
      call process%setup_terms ()

      call process_instance%init (process)
      call process%integrate (process_instance, 1, 1, 1000)
      call process%final_integration (1)

      call process_instance%setup_event_data ()
      call process_instance%init_simulation (1)
      call process%generate_weighted_event (process_instance, 1)
      call process_instance%evaluate_event_data ()

    end subroutine setup_testbed
```

FSR Shower

We generate a two-jet event and shower it using default settings for the WHIZARD
FSR shower.

⟨*Shower: execute tests*⟩≡

```
    call test (shower_1, "shower_1", &
        "disabled shower", &
        u, results)
```

⟨*Shower: tests*⟩+≡

```
    subroutine shower_1 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(process_library_t), target :: lib
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(process_t), target :: process
      type(process_instance_t), target :: process_instance
      integer :: factorization_mode
      logical :: keep_correlations
      class(evt_t), allocatable, target :: evt_trivial
      class(evt_t), allocatable, target :: evt_shower
      type(shower_settings_t) :: settings

      write (u, "(A)")  "* Test output: shower_1"
      write (u, "(A)")  "*   Purpose: Two-jet event with disabled shower"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize environment"
      write (u, "(A)")

      call syntax_model_file_init ()
      call setup_testbed (var_str ("shower_1"), &
          os_data, lib, model_list, model, process, process_instance)
```

```
write (u, "(A)")  "* Set up trivial transform"
write (u, "(A)")

allocate (evt_trivial_t :: evt_trivial)
call evt_trivial%connect (process_instance, model)
call evt_trivial%prepare_new_event (1, 1)
call evt_trivial%generate_unweighted ()

factorization_mode = FM_IGNORE_HELICITY
keep_correlations = .false.
call evt_trivial%make_particle_set (factorization_mode, keep_correlations)

select type (evt_trivial)
type is (evt_trivial_t)
   call evt_trivial%write (u)
   call write_separator_double (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Set up shower event transform"
write (u, "(A)")

settings%ps_fsr_active = .true.

select type (evt_shower)
type is (evt_shower_t)
   call evt_shower%init (settings, os_data)
end select

allocate (evt_shower_t :: evt_shower)
call evt_shower%connect (process_instance, model)
evt_trivial%next => evt_shower
evt_shower%previous => evt_trivial

call evt_shower%prepare_new_event (1, 1)
call evt_shower%generate_unweighted ()
call evt_shower%make_particle_set (factorization_mode, keep_correlations)

select type (evt_shower)
type is (evt_shower_t)
   call evt_shower%write (u)
   call write_separator_double (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call evt_shower%final ()
call evt_trivial%final ()
call process_instance%final ()
call process%final ()
call lib%final ()
call model_list%final ()
call syntax_model_file_final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: shower_1"

  end subroutine shower_1
```

Trivial Test

We generate a two-jet event and shower it using default settings. Since the
default settings have all

⟨*Shower: execute tests*⟩+≡
```
    call test (shower_2, "shower_2", &
        "final-state shower", &
        u, results)
```

⟨*Shower: tests*⟩+≡
```
  subroutine shower_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(process_library_t), target :: lib
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    type(process_t), target :: process
    type(process_instance_t), target :: process_instance
    integer :: factorization_mode
    logical :: keep_correlations
    class(evt_t), allocatable, target :: evt_trivial
    class(evt_t), allocatable, target :: evt_shower
    type(shower_settings_t) :: settings

    write (u, "(A)")  "* Test output: shower_2"
    write (u, "(A)")  "*   Purpose: Two-jet event with FSR shower"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize environment"
    write (u, "(A)")

    call syntax_model_file_init ()
    call setup_testbed (var_str ("shower_2"), &
        os_data, lib, model_list, model, process, process_instance)

    write (u, "(A)")  "* Set up trivial transform"
    write (u, "(A)")

    allocate (evt_trivial_t :: evt_trivial)
    call evt_trivial%connect (process_instance, model)
    call evt_trivial%prepare_new_event (1, 1)
    call evt_trivial%generate_unweighted ()

    factorization_mode = FM_IGNORE_HELICITY
    keep_correlations = .false.
    call evt_trivial%make_particle_set (factorization_mode, keep_correlations)
```

2415

```
      select type (evt_trivial)
      type is (evt_trivial_t)
         call evt_trivial%write (u)
         call write_separator_double (u)
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Set up shower event transform"
      write (u, "(A)")

      settings%ps_fsr_active = .true.

      allocate (evt_shower_t :: evt_shower)
      select type (evt_shower)
      type is (evt_shower_t)
         call evt_shower%init (settings, os_data)
      end select

      call evt_shower%connect (process_instance, model)
      evt_trivial%next => evt_shower
      evt_shower%previous => evt_trivial

      call evt_shower%prepare_new_event (1, 1)
      call evt_shower%generate_unweighted ()
      call evt_shower%make_particle_set (factorization_mode, keep_correlations)

      select type (evt_shower)
      type is (evt_shower_t)
         call evt_shower%write (u, testflag = .true.)
         call write_separator_double (u)
      end select

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call evt_shower%final ()
      call evt_trivial%final ()
      call process_instance%final ()
      call process%final ()
      call lib%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: shower_2"

    end subroutine shower_2
```

17.4   Complete Events

This module combines hard processes with decay chains, shower, and hadroniza-
tion (not implemented yet) to complete events. It also manages the input and

output of event records in various formats.

⟨events.f90⟩≡
  ⟨*File header*⟩

  `module events`

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    `use limits, only: FMT_12 !NODEP!`
    `use diagnostics !NODEP!`
    `use unit_tests`
    `use os_interface`

    `use ifiles`
    `use lexers`
    `use parser`

    `use subevents`
    `use variables`
    `use expressions`
    `use models`

    `use state_matrices`
    `use particles`
    `use interactions`
    `use subevt_expr`
    `use rng_base`
    `use process_libraries`
    `use processes`
    `use process_stacks`
    `use event_transforms`
    `use decays`
    `use shower`

  ⟨*Standard module head*⟩

  ⟨*Events: public*⟩

  ⟨*Events: parameters*⟩

  ⟨*Events: types*⟩

  ⟨*Events: interfaces*⟩

  `contains`

  ⟨*Events: procedures*⟩

  ⟨*Events: tests*⟩

  `end module events`

2417

17.4.1 Event configuration

The parameters govern the transformation of an event to a particle set.

The `safety_factor` reduces the acceptance probability for unweighting. If greater than one, excess events become less likely, but the reweighting efficiency also drops.

The `sigma` and `n` values, if nontrivial, allow for reweighting the events according to the requested `norm_mode`.

Various `parse_node_t` objects are taken from the SINDARIN input. They encode expressions that apply to the current event. The workspaces for evaluating those expressions are set up in the `event_expr_t` objects. Note that these are really pointers, so the actual nodes are not stored inside the event object.

⟨*Events: types*⟩≡
```
type :: event_config_t
   logical :: unweighted = .false.
   integer :: norm_mode = NORM_UNDEFINED
   integer :: factorization_mode = FM_IGNORE_HELICITY
   logical :: keep_correlations = .false.
   real(default) :: sigma = 1
   integer :: n = 1
   real(default) :: safety_factor = 1
   type(parse_node_t), pointer :: pn_selection => null ()
   type(parse_node_t), pointer :: pn_reweight => null ()
   type(parse_node_t), pointer :: pn_analysis => null ()
 contains
 ⟨Events: event config: TBP⟩
 end type event_config_t
```

Output.

⟨*Events: event config: TBP*⟩≡
```
procedure :: write => event_config_write
```

⟨*Events: procedures*⟩≡
```
subroutine event_config_write (object, unit, show_expressions)
  class(event_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_expressions
  integer :: u
  u = output_unit (unit)
  write (u, "(3x,A,L1)")  "Unweighted         = ", object%unweighted
  write (u, "(3x,A,A)")   "Normalization     = ", &
       char (event_normalization_string (object%norm_mode))
  write (u, "(3x,A)", advance="no")  "Helicity handling  = "
  select case (object%factorization_mode)
  case (FM_IGNORE_HELICITY)
     write (u, "(A)")  "drop"
  case (FM_SELECT_HELICITY)
     write (u, "(A)")  "select"
  case (FM_FACTOR_HELICITY)
     write (u, "(A)")  "factorize"
  end select
  write (u, "(3x,A,L1)")  "Keep correlations  = ", object%keep_correlations
  if (object%safety_factor /= 1) then
```

```
        write (u, "(3x,A," // FMT_12 // ")")  &
             "Safety factor      = ", object%safety_factor
      end if
    if (present (show_expressions)) then
       if (show_expressions) then
          if (associated (object%pn_selection)) then
             call write_separator (u)
             write (u, "(3x,A)") "Event selection expression:"
             call object%pn_selection%write (u)
          end if
          if (associated (object%pn_reweight)) then
             call write_separator (u)
             write (u, "(3x,A)") "Event reweighting expression:"
             call object%pn_reweight%write (u)
          end if
          if (associated (object%pn_analysis)) then
             call write_separator (u)
             write (u, "(3x,A)") "Analysis expression:"
             call object%pn_analysis%write (u)
          end if
       end if
    end if
  end subroutine event_config_write
```

17.4.2   Event normalization

The parameters for event normalization. For unweighted events, `NORM_UNIT` is intended as default, while for weighted events, it is `NORM_SIGMA`.

Note: the unit test for this is in `eio_data_2` below.

⟨*Events: parameters*⟩≡
```
    integer, parameter, public :: NORM_UNDEFINED = 0
    integer, parameter, public :: NORM_UNIT = 1
    integer, parameter, public :: NORM_N_EVT = 2
    integer, parameter, public :: NORM_SIGMA = 3
    integer, parameter, public :: NORM_S_N = 4
```

These functions translate between the user representation and the internal one.

⟨*Events: public*⟩≡
```
    public :: event_normalization_mode
    public :: event_normalization_string
```

⟨*Events: procedures*⟩+≡
```
    function event_normalization_mode (string, unweighted) result (mode)
      integer :: mode
      type(string_t), intent(in) :: string
      logical, intent(in) :: unweighted
      select case (lower_case (char (string)))
      case ("auto")
         if (unweighted) then
            mode = NORM_UNIT
         else
            mode = NORM_SIGMA
```

```
         end if
      case ("1")
         mode = NORM_UNIT
      case ("1/n")
         mode = NORM_N_EVT
      case ("sigma")
         mode = NORM_SIGMA
      case ("sigma/n")
         mode = NORM_S_N
      case default
         call msg_fatal ("Event normalization: unknown value '" &
              // char (string) // "'")
      end select
   end function event_normalization_mode

   function event_normalization_string (norm_mode) result (string)
     integer, intent(in) :: norm_mode
     type(string_t) :: string
     select case (norm_mode)
     case (NORM_UNDEFINED); string = "[undefined]"
     case (NORM_UNIT);      string = "'1'"
     case (NORM_N_EVT);     string = "'1/n'"
     case (NORM_SIGMA);     string = "'sigma'"
     case (NORM_S_N);       string = "'sigma/n'"
     case default;          string = "???"
     end select
   end function event_normalization_string
```

We place this here as a generic helper, so we can update event weights whenever we need, not just in connection with an event sample data object.

⟨*Events: public*⟩+≡
```
    public :: event_normalization_update
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_normalization_update (weight, sigma, n, mode_new, mode_old)
      real(default), intent(inout) :: weight
      real(default), intent(in) :: sigma
      integer, intent(in) :: n
      integer, intent(in) :: mode_new, mode_old
      if (mode_new /= mode_old) then
         if (sigma > 0 .and. n > 0) then
            weight = weight / factor (mode_old) * factor (mode_new)
         else
            call msg_fatal ("Event normalization update: null sample")
         end if
      end if
    contains
      function factor (mode)
        real(default) :: factor
        integer, intent(in) :: mode
        select case (mode)
        case (NORM_UNIT);  factor = 1._default
        case (NORM_N_EVT); factor = 1._default / n
        case (NORM_SIGMA); factor = sigma
```

```
      case (NORM_S_N);    factor = sigma / n
      case default
         call msg_fatal ("Event normalization update: undefined mode")
      end select
    end function factor
  end subroutine event_normalization_update
```

17.4.3 The event type

Each event has a single elementary process as its core. This process may be dressed by a shower, a decay chain etc. For this core we implement pointers to a process instance.

The actual (dressed) event record is implemented as a `particle_set` object.

A list of event transforms (class `evt_t`) transform the connected interactions of the process instance into the final particle set. In this list, the first transform is always the trivial one, which just factorizes the process instance. Subsequent transforms may apply decays, etc. The `particle_set` pointer identifies the particle set that we want to be analyzed and returned by the event, usually the last one.

Squared matrix element and weight values: when reading events from file, the `ref` value is the number in the file, while the `prc` value is the number that we calculate from the momenta in the file, possibly with different parameters. When generating events the first time, or if we do not recalculate, the numbers should coincide. Furthermore, the array of `alt` values is copied from an array of alternative event records. These values should represent calculated values.

The `sqme` and `weight` values mirror corresponding values in the `expr` subobject. The idea is that when generating or reading events, the event record is filled first, then the `expr` object acquires copies. These copies are used for writing events and as targets for pointer variables in the analysis expression.

All data that involve user-provided expressions (selection, reweighting, analysis) are handled by the `expr` subobject. In particular, evaluating the event-selection expression sets the `passed` flag. Furthermore, the `expr` subobject collects data that can be used in the analysis and should be written to file, including copies of `sqme` and `weight`.

⟨*Events: public*⟩+≡
```
    public :: event_t
```

⟨*Events: types*⟩+≡
```
    type :: event_t
      type(event_config_t) :: config
      type(process_t), pointer :: process => null ()
      type(process_instance_t), pointer :: instance => null ()
      class(rng_t), allocatable :: rng
      integer :: selected_i_mci = 0
      integer :: selected_i_term = 0
      integer :: selected_channel = 0
      logical :: is_complete = .false.
      class(evt_t), pointer :: transform_first => null ()
      class(evt_t), pointer :: transform_last => null ()
      logical :: particle_set_exists = .false.
      type(particle_set_t), pointer :: particle_set => null ()
```

```
      logical :: sqme_ref_is_known = .false.
      real(default) :: sqme_ref = 0
      logical :: sqme_prc_is_known = .false.
      real(default) :: sqme_prc = 0
      logical :: weight_ref_is_known = .false.
      real(default) :: weight_ref = 0
      logical :: weight_prc_is_known = .false.
      real(default) :: weight_prc = 0
      logical :: excess_prc_is_known = .false.
      real(default) :: excess_prc = 0
      integer :: n_alt = 0
      logical :: sqme_alt_is_known = .false.
      real(default), dimension(:), allocatable :: sqme_alt
      logical :: weight_alt_is_known = .false.
      real(default), dimension(:), allocatable :: weight_alt
      type(event_expr_t) :: expr
      logical :: selection_evaluated = .false.
      logical :: passed = .false.
      real(default) :: reweight = 1
      logical :: analysis_flag = .false.
    contains
    ⟨Events: event: TBP⟩
    end type event_t
```

Finalizer: the list of event transforms is deleted iteratively.

⟨*Events: event: TBP*⟩≡
```
      procedure :: final => event_final
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_final (object)
      class(event_t), intent(inout) :: object
      class(evt_t), pointer :: evt
      if (allocated (object%rng))  call object%rng%final ()
      call object%expr%final ()
      do while (associated (object%transform_first))
         evt => object%transform_first
         object%transform_first => evt%next
         call evt%final ()
         deallocate (evt)
      end do
    end subroutine event_final
```

Output.

Particle set: this is a pointer to one of the event transforms, so it should suffice to print the latter.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: write => event_write
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_write (object, unit, show_process, show_transforms, &
        show_decay, verbose, testflag)
      class(event_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: show_process, show_transforms, show_decay
```

```
logical, intent(in), optional :: verbose
logical, intent(in), optional :: testflag
logical :: prc, trans, dec, verb
class(evt_t), pointer :: evt
integer :: u, i
u = output_unit (unit)
prc = .true.;  if (present (show_process))  prc = show_process
trans = .true.;  if (present (show_transforms))  trans = show_transforms
dec = .true.;  if (present (show_decay))  dec = show_decay
verb = .false.;  if (present (verbose))  verb = verbose
call write_separator_double (u)
if (object%is_complete) then
   write (u, "(1x,A)")  "Event"
else
   write (u, "(1x,A)")  "Event [incomplete]"
end if
call write_separator (u)
call object%config%write (u)
if (object%sqme_ref_is_known .or. object%weight_ref_is_known) then
   call write_separator (u)
end if
if (object%sqme_ref_is_known) then
   write (u, "(3x,A,ES19.12)")  "Squared matrix el. = ", object%sqme_ref
   if (object%sqme_alt_is_known) then
      do i = 1, object%n_alt
         write (u, "(5x,A,ES19.12,1x,I0)")  &
              "alternate sqme  = ", object%sqme_alt(i), i
      end do
   end if
end if
if (object%weight_ref_is_known) then
   write (u, "(3x,A,ES19.12)")  "Event weight       = ", object%weight_ref
   if (object%weight_alt_is_known) then
      do i = 1, object%n_alt
         write (u, "(5x,A,ES19.12,1x,I0)")  &
              "alternate weight = ", object%weight_alt(i), i
      end do
   end if
end if
if (object%selected_i_mci /= 0) then
   call write_separator (u)
   write (u, "(3x,A,I0)")  "Selected MCI group = ", object%selected_i_mci
   write (u, "(3x,A,I0)")  "Selected term      = ", object%selected_i_term
   write (u, "(3x,A,I0)")  "Selected channel   = ", object%selected_channel
end if
if (object%selection_evaluated) then
   call write_separator (u)
   write (u, "(3x,A,L1)")  "Passed selection   = ", object%passed
   if (object%passed) then
      write (u, "(3x,A,ES19.12)") &
           "Reweighting factor = ", object%reweight
      write (u, "(3x,A,L1)") &
           "Analysis flag      = ", object%analysis_flag
   end if
```

```
         end if
         if (associated (object%instance)) then
            if (prc) then
               if (verb) then
                  call object%instance%write (u, testflag)
               else
                  call object%instance%write_header (u)
               end if
            end if
            if (trans) then
               evt => object%transform_first
               do while (associated (evt))
                  select type (evt)
                  type is (evt_trivial_t)
                     call evt%write (u, verbose = verbose, testflag = testflag)
                  type is (evt_decay_t)
                     call evt%write (u, show_decay_tree = dec, &
                           show_processes = dec .and. verb, verbose = verb)
                  type is (evt_shower_t)
                     call evt%write (u)
                  end select
                  call write_separator_double (u)
                  evt => evt%next
               end do
            else
               call write_separator_double (u)
            end if
            if (object%expr%subevt_filled) then
               call object%expr%write (u)
               call write_separator_double (u)
            end if
         else
            call write_separator_double (u)
            write (u, "(1x,A)")  "Process instance: [undefined]"
            call write_separator_double (u)
         end if
      end subroutine event_write
```

17.4.4 Initialization

Initialize: set configuration parameters, using a variable list. We do not call this **init**, because this method name will be used by a type extension.

The default normalization is `NORM_SIGMA`, since the default generation mode is weighted.

For unweighted events, we may want to a apply a safety factor to event rejection. (By default, this factor is unity and can be ignored.)

We also allocate the trivial event transform, which is always the first one.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: basic_init => event_init
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_init (event, var_list, n_alt)
```

```
      class(event_t), intent(out) :: event
      type(var_list_t), intent(in), optional :: var_list
      integer, intent(in), optional :: n_alt
      type(string_t) :: norm_string
      logical :: polarized_events
      if (present (var_list)) then
         event%config%unweighted = var_list_get_lval (var_list, &
              var_str ("?unweighted"))
         norm_string = var_list_get_sval (var_list, &
              var_str ("$sample_normalization"))
         event%config%norm_mode = &
              event_normalization_mode (norm_string, event%config%unweighted)
         polarized_events = &
              var_list_get_lval (var_list, var_str ("?polarized_events"))
         if (polarized_events) then
            event%config%factorization_mode = FM_SELECT_HELICITY
         else
            event%config%factorization_mode = FM_IGNORE_HELICITY
         end if
         if (event%config%unweighted) then
            event%config%safety_factor = var_list_get_rval (var_list, &
                 var_str ("safety_factor"))
         end if
      else
         event%config%norm_mode = NORM_SIGMA
      end if
      if (present (n_alt)) then
         event%n_alt = n_alt
         allocate (event%sqme_alt (n_alt))
         allocate (event%weight_alt (n_alt))
         call event%expr%init (n_alt)
      end if
      allocate (evt_trivial_t :: event%transform_first)
      event%transform_last => event%transform_first
    end subroutine event_init
```

Set the `sigma` and `n` values in the configuration record that determine non-standard event normalizations. If these numbers are not set explicitly, the default value for both is unity, and event renormalization has no effect.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: set_sigma => event_set_sigma
    procedure :: set_n => event_set_n
```

⟨*Events: procedures*⟩+≡
```
    elemental subroutine event_set_sigma (event, sigma)
      class(event_t), intent(inout) :: event
      real(default), intent(in) :: sigma
      event%config%sigma = sigma
    end subroutine event_set_sigma

    elemental subroutine event_set_n (event, n)
      class(event_t), intent(inout) :: event
      integer, intent(in) :: n
      event%config%n = n
```

```
      end subroutine event_set_n
```

Append an event transform (decays, etc.). The transform is not yet connected to a process. The transform is then considered to belong to the event object, and will be finalized together with it. The original pointer is removed.

We can assume that the trivial transform is already present in the event object, at least.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: import_transform => event_import_transform
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_import_transform (event, evt)
      class(event_t), intent(inout) :: event
      class(evt_t), intent(inout), pointer :: evt
      event%transform_last%next => evt
      evt%previous => event%transform_last
      event%transform_last => evt
      evt => null ()
    end subroutine event_import_transform
```

We link the event to an existing process instance. This includes the variable list, which is linked to the process variable list. Note that this is not necessarily identical to the variable list used for event initialization.

The variable list will contain pointers to **event** subobjects, therefore the **target** attribute.

Once we have a process connected, we can use it to obtain an event generator instance.

The model and process stack may be needed by event transforms. The current model setting may be different from the model in the process (regarding unstable particles, etc.). The process stack can be used for assigning extra processes that we need for the event transforms.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: connect => event_connect
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_connect (event, process_instance, model, process_stack)
      class(event_t), intent(inout), target :: event
      type(process_instance_t), intent(in), target :: process_instance
      type(model_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      type(string_t) :: id
      integer :: num_id
      class(evt_t), pointer :: evt
      event%process => process_instance%process
      event%instance => process_instance
      id = event%process%get_id ()
      if (id /= "")  call event%expr%set_process_id (id)
      num_id = event%process%get_num_id ()
      if (num_id /= 0)  call event%expr%set_process_num_id (num_id)
      call event%expr%setup_vars (event%process%get_sqrts ())
      call event%expr%link_var_list (event%process%get_var_list_ptr ())
      call event%process%make_rng (event%rng)
```

```
      evt => event%transform_first
      do while (associated (evt))
          call evt%connect (process_instance, model, process_stack)
          evt => evt%next
      end do
   end subroutine event_connect
```

Set the parse nodes for the associated expressions, individually. The parse-node
pointers may be null.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: set_selection => event_set_selection
    procedure :: set_reweight => event_set_reweight
    procedure :: set_analysis => event_set_analysis
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_set_selection (event, pn_selection)
      class(event_t), intent(inout) :: event
      type(parse_node_t), intent(in), pointer :: pn_selection
      event%config%pn_selection => pn_selection
    end subroutine event_set_selection

    subroutine event_set_reweight (event, pn_reweight)
      class(event_t), intent(inout) :: event
      type(parse_node_t), intent(in), pointer :: pn_reweight
      event%config%pn_reweight => pn_reweight
    end subroutine event_set_reweight

    subroutine event_set_analysis (event, pn_analysis)
      class(event_t), intent(inout) :: event
      type(parse_node_t), intent(in), pointer :: pn_analysis
      event%config%pn_analysis => pn_analysis
    end subroutine event_set_analysis
```

Create evaluation trees from the parse trees. The `target` attribute is required
because the expressions contain pointers to event subobjects.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: setup_expressions => event_setup_expressions
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_setup_expressions (event)
      class(event_t), intent(inout), target :: event
      call event%expr%setup_selection (event%config%pn_selection)
      call event%expr%setup_analysis (event%config%pn_analysis)
      call event%expr%setup_reweight (event%config%pn_reweight)
    end subroutine event_setup_expressions
```

17.4.5 Evaluation

To fill the `particle_set`, i.e., the event record proper, we have to apply all
event transforms in order. The last transform should fill its associated particle
set, factorizing the state matrix according to the current settings. There are
several parameters in the event configuration that control this.

We always fill the particle set for the first transform (the hard process) and the last transform, if different from the first (the fully dressed process).

Each event transform is an event generator of its own. We choose to generate an *unweighted* event for each of them, even if the master event is assumed to be weighted. Thus, the overall event weight is the one of the hard process only. (There may be more options in future extensions.)

We can generate the two random numbers that the factorization needs. For testing purpose, we allow for providing them explicitly, as an option.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: evaluate_transforms => event_evaluate_transforms
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_evaluate_transforms (event, r)
      class(event_t), intent(inout) :: event
      real(default), dimension(:), intent(in), optional :: r
      class(evt_t), pointer :: evt
      integer :: i_term
      event%particle_set_exists = .false.
      call event%check ()
      if (event%instance%is_complete_event ()) then
         call event%instance%select_i_term (i_term)
         event%selected_i_term = i_term
         evt => event%transform_first
         do while (associated (evt))
            call evt%prepare_new_event &
                 (event%selected_i_mci, event%selected_i_term)
            evt => evt%next
         end do
         evt => event%transform_first
         do while (associated (evt))
            call evt%generate_unweighted ()
            if (signal_is_pending ())  return
            call evt%make_particle_set (event%config%factorization_mode, &
                 event%config%keep_correlations)
            if (signal_is_pending ())  return
            if (.not. evt%particle_set_exists)  exit
            evt => evt%next
         end do
         evt => event%transform_last
         if (associated (evt) .and. evt%particle_set_exists) then
            event%particle_set => evt%particle_set
            call event%accept_particle_set ()
         end if
      end if
    end subroutine event_evaluate_transforms
```

Evaluate the event-related expressions, given a valid `particle_set`. If `update_sqme` is set, we use the process instance for the `sqme_prc` value. The `sqme_ref` value is always taken from the event record.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: evaluate_expressions => event_evaluate_expressions
```

⟨*Events: procedures*⟩+≡

```
subroutine event_evaluate_expressions (event)
  class(event_t), intent(inout) :: event
  if (event%particle_set_exists) then
     call event%expr%fill_subevt (event%particle_set)
  end if
  if (event%weight_ref_is_known) then
     call event%expr%set (weight_ref = event%weight_ref)
  end if
  if (event%weight_prc_is_known) then
     call event%expr%set (weight_prc = event%weight_prc)
  end if
  if (event%excess_prc_is_known) then
     call event%expr%set (excess_prc = event%excess_prc)
  end if
  if (event%sqme_ref_is_known) then
     call event%expr%set (sqme_ref = event%sqme_ref)
  end if
  if (event%sqme_prc_is_known) then
     call event%expr%set (sqme_prc = event%sqme_prc)
  end if
  if (event%particle_set_exists) then
     call event%expr%evaluate &
          (event%passed, event%reweight, event%analysis_flag)
     event%selection_evaluated = .true.
  end if
end subroutine event_evaluate_expressions
```

Set alternate sqme and weight arrays. This should be merged with the previous routine, if the expressions are allowed to refer to these values.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: store_alt_values => event_store_alt_values
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_store_alt_values (event)
      class(event_t), intent(inout) :: event
      if (event%weight_alt_is_known) then
         call event%expr%set (weight_alt = event%weight_alt)
      end if
      if (event%sqme_alt_is_known) then
         call event%expr%set (sqme_alt = event%sqme_alt)
      end if
    end subroutine event_store_alt_values
```

17.4.6 Reset to empty state

Applying this, current event contents are marked as incomplete but are not deleted. In particular, the initialization is kept.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: reset => event_reset
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_reset (event)
```

```
     class(event_t), intent(inout) :: event
     class(evt_t), pointer :: evt
     event%selected_i_mci = 0
     event%selected_i_term = 0
     event%selected_channel = 0
     event%is_complete = .false.
     event%particle_set_exists = .false.
     event%sqme_ref_is_known = .false.
     event%sqme_prc_is_known = .false.
     event%sqme_alt_is_known = .false.
     event%weight_ref_is_known = .false.
     event%weight_prc_is_known = .false.
     event%weight_alt_is_known = .false.
     event%excess_prc_is_known = .false.
     call event%expr%reset ()
     event%selection_evaluated = .false.
     event%passed = .false.
     event%analysis_flag = .false.
     if (associated (event%instance)) then
        call event%instance%reset (reset_mci = .true.)
     end if
     evt => event%transform_first
     do while (associated (evt))
        call evt%reset ()
        evt => evt%next
     end do
  end subroutine event_reset
```

17.4.7   Squared Matrix Element and Weight

Set the appropriate entry directly.

⟨*Events: event: TBP*⟩+≡
```
     procedure :: set => event_set
```

⟨*Events: procedures*⟩+≡
```
     subroutine event_set (event, &
         weight_ref, weight_prc, weight_alt, &
         excess_prc, &
         sqme_ref, sqme_prc, sqme_alt)
       class(event_t), intent(inout) :: event
       real(default), intent(in), optional :: weight_ref, weight_prc
       real(default), intent(in), optional :: sqme_ref, sqme_prc
       real(default), dimension(:), intent(in), optional :: sqme_alt, weight_alt
       real(default), intent(in), optional :: excess_prc
       if (present (sqme_ref)) then
          event%sqme_ref_is_known = .true.
          event%sqme_ref = sqme_ref
       end if
       if (present (sqme_prc)) then
          event%sqme_prc_is_known = .true.
          event%sqme_prc = sqme_prc
       end if
       if (present (sqme_alt)) then
```

```
            event%sqme_alt_is_known = .true.
            event%sqme_alt = sqme_alt
         end if
         if (present (weight_ref)) then
            event%weight_ref_is_known = .true.
            event%weight_ref = weight_ref
         end if
         if (present (weight_prc)) then
            event%weight_prc_is_known = .true.
            event%weight_prc = weight_prc
         end if
         if (present (weight_alt)) then
            event%weight_alt_is_known = .true.
            event%weight_alt = weight_alt
         end if
         if (present (excess_prc)) then
            event%excess_prc_is_known = .true.
            event%excess_prc = excess_prc
         end if
      end subroutine event_set
```

Transfer the result of the process instance calculation to the event record header.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: import_instance_results => event_import_instance_results
```

⟨*Events: procedures*⟩+≡
```
      subroutine event_import_instance_results (event)
        class(event_t), intent(inout) :: event
        if (associated (event%instance)) then
           if (event%instance%has_evaluated_trace ()) then
              call event%set ( &
                   sqme_prc = event%instance%get_sqme (), &
                   weight_prc = event%instance%get_weight (), &
                   excess_prc = event%instance%get_excess () &
                   )
           end if
        end if
      end subroutine event_import_instance_results
```

Duplicate the instance result / the reference result in the event record.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: accept_sqme_ref => event_accept_sqme_ref
      procedure :: accept_sqme_prc => event_accept_sqme_prc
      procedure :: accept_weight_ref => event_accept_weight_ref
      procedure :: accept_weight_prc => event_accept_weight_prc
```

⟨*Events: procedures*⟩+≡
```
      subroutine event_accept_sqme_ref (event)
        class(event_t), intent(inout) :: event
        if (event%sqme_ref_is_known) then
           call event%set (sqme_prc = event%sqme_ref)
        end if
      end subroutine event_accept_sqme_ref
```

```
      subroutine event_accept_sqme_prc (event)
        class(event_t), intent(inout) :: event
        if (event%sqme_prc_is_known) then
           call event%set (sqme_ref = event%sqme_prc)
        end if
      end subroutine event_accept_sqme_prc

      subroutine event_accept_weight_ref (event)
        class(event_t), intent(inout) :: event
        if (event%weight_ref_is_known) then
           call event%set (weight_prc = event%weight_ref)
        end if
      end subroutine event_accept_weight_ref

      subroutine event_accept_weight_prc (event)
        class(event_t), intent(inout) :: event
        if (event%weight_prc_is_known) then
           call event%set (weight_ref = event%weight_prc)
        end if
      end subroutine event_accept_weight_prc
```

Update the weight normalization, just after generation. Unweighted and weighted
events are generated with a different default normalization. The intended nor-
malization is stored in the configuration record.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: update_normalization => event_update_normalization
```

⟨*Events: procedures*⟩+≡
```
      subroutine event_update_normalization (event, mode_ref)
        class(event_t), intent(inout) :: event
        integer, intent(in), optional :: mode_ref
        integer :: mode_old
        if (present (mode_ref)) then
           mode_old = mode_ref
        else if (event%config%unweighted) then
           mode_old = NORM_UNIT
        else
           mode_old = NORM_SIGMA
        end if
        call event_normalization_update (event%weight_prc, &
             event%config%sigma, event%config%n, &
             mode_new = event%config%norm_mode, &
             mode_old = mode_old)
        call event_normalization_update (event%excess_prc, &
             event%config%sigma, event%config%n, &
             mode_new = event%config%norm_mode, &
             mode_old = mode_old)
      end subroutine event_update_normalization
```

The event is complete if it has a particle set plus valid entries for the sqme and
weight values.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: check => event_check
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_check (event)
      class(event_t), intent(inout) :: event
      event%is_complete = event%particle_set_exists &
           .and. event%sqme_ref_is_known &
           .and. event%sqme_prc_is_known &
           .and. event%weight_ref_is_known &
           .and. event%weight_prc_is_known
      if (event%n_alt /= 0) then
         event%is_complete = event%is_complete &
              .and. event%sqme_alt_is_known &
              .and. event%weight_alt_is_known
      end if
    end subroutine event_check
```

17.4.8   Generation

Assuming that we have a valid process associated to the event, we generate an event. We complete the event data, then factorize the spin density matrix and transfer it to the particle set.

When done, we retrieve squared matrix element and weight. In case of explicit generation, the reference values coincide with the process values, so we `accept` the latter.

The explicit random number argument `r` should be generated by a random-number generator. It is taken for the factorization algorithm, bypassing the event-specific random-number generator. This is useful for deterministic testing.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: generate => event_generate
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_generate (event, i_mci, r)
      class(event_t), intent(inout) :: event
      integer, intent(in) :: i_mci
      real(default), dimension(:), intent(in), optional :: r
      call event%reset ()
      event%selected_i_mci = i_mci
      if (event%config%unweighted) then
         call event%process%generate_unweighted_event (event%instance, i_mci)
         if (signal_is_pending ()) return
         call event%instance%evaluate_event_data ()
         call event%instance%normalize_weight ()
      else
         call event%process%generate_weighted_event (event%instance, i_mci)
         if (signal_is_pending ()) return
         call event%instance%evaluate_event_data ()
      end if
      event%selected_channel = event%instance%get_channel ()
      call event%import_instance_results ()
      call event%accept_sqme_prc ()
      call event%update_normalization ()
      call event%accept_weight_prc ()
      call event%evaluate_transforms (r)
```

2433

```
      if (signal_is_pending ())  return
      call event%check ()
   end subroutine event_generate
```

Get a copy of the particle set belonging to the hard process.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: get_particle_set_hard_proc => event_get_particle_set_hard_proc
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_get_particle_set_hard_proc (event, pset)
      class(event_t), intent(in) :: event
      type(particle_set_t), intent(out) :: pset
      class(evt_t), pointer :: evt
      evt => event%transform_first
      pset = evt%particle_set
    end subroutine event_get_particle_set_hard_proc
```

17.4.9   Recovering an event

Select MC group, term, and integration channel.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: select => event_select
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_select (event, i_mci, i_term, channel)
      class(event_t), intent(inout) :: event
      integer, intent(in) :: i_mci, i_term, channel
      if (associated (event%instance)) then
         event%selected_i_mci = i_mci
         event%selected_i_term = i_term
         event%selected_channel = channel
      else
         call msg_bug ("Event: select term: process instance undefined")
      end if
    end subroutine event_select
```

Copy a particle set into the event record.

We deliberately use the first (the trivial) transform for this, i.e., the hard process. Thus, with the current implementation, event readers can only be used for reading hard events. However, once we have reconstructed the master process, we may apply all event transforms to generate a fully dressed event.

Since this makes all subsequent event transforms invalid, we call `reset` on them.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: set_particle_set_hard_proc => event_set_particle_set_hard_proc
```

⟨*Events: procedures*⟩+≡
```
    subroutine event_set_particle_set_hard_proc (event, particle_set)
      class(event_t), intent(inout) :: event
      type(particle_set_t), intent(in) :: particle_set
      class(evt_t), pointer :: evt
      evt => event%transform_first
```

```
         call evt%set_particle_set (particle_set, &
              event%selected_i_mci, event%selected_i_term)
         event%particle_set => evt%particle_set
         call event%accept_particle_set ()
         evt => evt%next
         do while (associated (evt))
            call evt%reset ()
            evt => evt%next
         end do
      end subroutine event_set_particle_set_hard_proc
```

Mark the particle set as existing, assuming that it has just been filled somehow.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: accept_particle_set => event_accept_particle_set
```

⟨*Events: procedures*⟩+≡
```
      subroutine event_accept_particle_set (event)
        class(event_t), intent(inout) :: event
        event%particle_set_exists = .true.
      end subroutine event_accept_particle_set
```

Here we try to recover an event from the `particle_set` subobject and recalculate the structure functions and matrix elements. We have the appropriate `process` object and an initialized `process_instance` at hand, so beam and configuration data are known. From the `particle_set`, we get the momenta.

The quantum-number information may be incomplete, e.g., helicity information may be partial or absent. We recover the event just from the momentum configuration.

We do not transfer the matrix element from the process instance to the event record, as we do when generating an event. The event record may contain the matrix element as read from file, and the current calculation may use different parameters. We thus can compare old and new values.

The event `weight` may also be known already. If yes, we pass it to the `evaluate_event_data` procedure. It should already be normalized. If we have an `weight_factor` value, we obtain the event weight by multiplying the computed `sqme` by this factor. Otherwise, we make use of the MCI setup (which should be valid then) to compute the event weight, and we should normalize the result just as when generating events.

Evaluating event expressions must also be done separately.

⟨*Events: event: TBP*⟩+≡
```
      procedure :: recalculate => event_recalculate
```

⟨*Events: procedures*⟩+≡
```
      subroutine event_recalculate &
           (event, update_sqme, weight_factor, recover_beams)
        class(event_t), intent(inout) :: event
        logical, intent(in) :: update_sqme
        real(default), intent(in), optional :: weight_factor
        logical, intent(in), optional :: recover_beams
        integer :: i_mci, i_term, channel
        if (event%particle_set_exists) then
           i_mci = event%selected_i_mci
```

```
            i_term = event%selected_i_term
            channel = event%selected_channel
            if (i_mci == 0 .or. i_term == 0 .or. channel == 0) then
                call msg_bug ("Event: recalculate: undefined selection parameters")
            end if
            call event%instance%choose_mci (i_mci)
            call event%instance%set_trace (event%particle_set, i_term, recover_beams)
            call event%instance%recover (channel, i_term, update_sqme)
            if (signal_is_pending ())  return
            if (update_sqme .and. present (weight_factor)) then
                call event%instance%evaluate_event_data &
                     (weight = event%instance%get_sqme () * weight_factor)
            else if (event%weight_ref_is_known) then
                call event%instance%evaluate_event_data &
                     (weight = event%weight_ref)
            else
                call event%process%recover_event (event%instance, i_term)
                if (signal_is_pending ())  return
                call event%instance%evaluate_event_data ()
                if (event%config%unweighted) then
                    call event%instance%normalize_weight ()
                end if
            end if
            if (signal_is_pending ())  return
            if (update_sqme) then
                call event%import_instance_results ()
            else
                call event%accept_sqme_ref ()
                call event%accept_weight_ref ()
            end if
        else
            call msg_bug ("Event: can't recalculate, particle set is undefined")
        end if
    end subroutine event_recalculate
```

17.4.10   Access content

Pointer to the associated process object.

⟨*Events: event: TBP*⟩+≡
```
    procedure :: get_process_ptr => event_get_process_ptr
```

⟨*Events: procedures*⟩+≡
```
    function event_get_process_ptr (event) result (ptr)
      class(event_t), intent(in) :: event
      type(process_t), pointer :: ptr
      ptr => event%process
    end function event_get_process_ptr
```

Tell whether the particle set is defined:

⟨*Events: event: TBP*⟩+≡
```
    procedure :: has_particle_set => event_has_particle_set
```

⟨Events: procedures⟩+≡
```
    function event_has_particle_set (event) result (flag)
      class(event_t), intent(in) :: event
      logical :: flag
      flag = event%particle_set_exists
    end function event_has_particle_set
```

Pointer to the particle set.

⟨Events: event: TBP⟩+≡
```
    procedure :: get_particle_set_ptr => event_get_particle_set_ptr
```

⟨Events: procedures⟩+≡
```
    function event_get_particle_set_ptr (event) result (ptr)
      class(event_t), intent(in), target :: event
      type(particle_set_t), pointer :: ptr
      ptr => event%particle_set
    end function event_get_particle_set_ptr
```

Return the current values of indices: the MCI group of components, the term index (different terms corresponding, potentially, to different effective kinematics), and the MC integration channel. The `i_mci` call is delegated to the current process instance.

⟨Events: event: TBP⟩+≡
```
    procedure :: get_i_mci => event_get_i_mci
    procedure :: get_i_term => event_get_i_term
    procedure :: get_channel => event_get_channel
```

⟨Events: procedures⟩+≡
```
    function event_get_i_mci (event) result (i_mci)
      class(event_t), intent(in) :: event
      integer :: i_mci
      i_mci = event%selected_i_mci
    end function event_get_i_mci

    function event_get_i_term (event) result (i_term)
      class(event_t), intent(in) :: event
      integer :: i_term
      i_term = event%selected_i_term
    end function event_get_i_term

    function event_get_channel (event) result (channel)
      class(event_t), intent(in) :: event
      integer :: channel
      channel = event%selected_channel
    end function event_get_channel
```

Return the currently selected normalization mode, or alternate normalization mode.

⟨Events: event: TBP⟩+≡
```
    procedure :: get_norm_mode => event_get_norm_mode
```

⟨*Events: procedures*⟩+≡
```
elemental function event_get_norm_mode (event) result (norm_mode)
  class(event_t), intent(in) :: event
  integer :: norm_mode
  norm_mode = event%config%norm_mode
end function event_get_norm_mode
```

Return the kinematical weight, defined as the ratio of event weight and squared matrix element.

⟨*Events: event: TBP*⟩+≡
```
procedure :: get_kinematical_weight => event_get_kinematical_weight
```

⟨*Events: procedures*⟩+≡
```
function event_get_kinematical_weight (event) result (f)
  class(event_t), intent(in) :: event
  real(default) :: f
  if (event%sqme_ref_is_known .and. event%weight_ref_is_known &
       .and. event%sqme_ref /= 0) then
     f = event%weight_ref / event%sqme_ref
  else
     f = 0
  end if
end function event_get_kinematical_weight
```

Return data used by external event formats.

⟨*Events: event: TBP*⟩+≡
```
procedure :: get_fac_scale => event_get_fac_scale
procedure :: get_alpha_s => event_get_alpha_s
```

⟨*Events: procedures*⟩+≡
```
function event_get_fac_scale (event) result (fac_scale)
  class(event_t), intent(in) :: event
  real(default) :: fac_scale
  fac_scale = event%instance%get_fac_scale (event%selected_i_term)
end function event_get_fac_scale

function event_get_alpha_s (event) result (alpha_s)
  class(event_t), intent(in) :: event
  real(default) :: alpha_s
  alpha_s = event%instance%get_alpha_s (event%selected_i_term)
end function event_get_alpha_s
```

Eliminate numerical noise in the `subevt` expression and in the event transforms (which includes associated process instances).

⟨*Events: public*⟩+≡
```
public :: pacify
```

⟨*Events: interfaces*⟩≡
```
interface pacify
   module procedure pacify_event
end interface pacify
```

⟨*Events: procedures*⟩+≡

```
subroutine pacify_event (event)
  class(event_t), intent(inout) :: event
  class(evt_t), pointer :: evt
  if (event%particle_set_exists)  call pacify (event%particle_set)
  if (event%expr%subevt_filled)  call pacify (event%expr)
  evt => event%transform_first
  do while (associated (evt))
     select type (evt)
     type is (evt_decay_t);  call pacify (evt)
     end select
     evt => evt%next
  end do
end subroutine pacify_event
```

17.4.11   Unit tests

⟨*Events: public*⟩+≡

```
public :: events_test
```

⟨*Events: tests*⟩≡

```
subroutine events_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*Events: execute tests*⟩
```
end subroutine events_test
```

Empty event record

⟨*Events: execute tests*⟩≡

```
call test (events_1, "events_1", &
     "empty event record", &
     u, results)
```

⟨*Events: tests*⟩+≡

```
subroutine events_1 (u)
  integer, intent(in) :: u
  type(event_t), target :: event

  write (u, "(A)")  "* Test output: events_1"
  write (u, "(A)")  "*   Purpose: display an empty event object"
  write (u, "(A)")

  call event%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: events_1"

end subroutine events_1
```

Simple event

⟨*Events: execute tests*⟩+≡
```
call test (events_2, "events_2", &
     "generate event", &
     u, results)
```

⟨*Events: tests*⟩+≡
```
subroutine events_2 (u)
  integer, intent(in) :: u
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_instance_t), allocatable, target :: process_instance
  type(model_list_t) :: model_list

  write (u, "(A)")  "* Test output: events_2"
  write (u, "(A)")  "*   Purpose: generate and display an event"
  write (u, "(A)")

  call syntax_model_file_init ()

  write (u, "(A)")  "* Generate test process event"

  allocate (process)
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model_list)
  call process_instance%setup_event_data ()

  write (u, "(A)")
  write (u, "(A)")  "* Initialize event object"

  allocate (event)
  call event%basic_init ()
  call event%connect (process_instance, process%get_model_ptr ())

  write (u, "(A)")
  write (u, "(A)")  "* Generate test process event"

  call process%generate_weighted_event (process_instance, 1)

  write (u, "(A)")
  write (u, "(A)")  "* Fill event object"
  write (u, "(A)")

  call event%generate (1, [0.4_default, 0.4_default])
  call event%evaluate_expressions ()
  call event%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call event%final ()
  deallocate (event)

  call cleanup_test_process (process, process_instance)
```

2440

```
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: events_2"

    end subroutine events_2
```

Event expressions

After generating an event, fill the `subevt` and evaluate expressions for selection,
reweighting, and analysis.

⟨*Events: execute tests*⟩+≡
```
    call test (events_3, "events_3", &
         "expression evaluation", &
         u, results)
```

⟨*Events: tests*⟩+≡
```
    subroutine events_3 (u)
      integer, intent(in) :: u
      type(string_t) :: expr_text
      type(ifile_t) :: ifile
      type(stream_t) :: stream
      type(parse_tree_t) :: pt_selection, pt_reweight, pt_analysis
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(model_list_t) :: model_list

      write (u, "(A)")  "* Test output: events_3"
      write (u, "(A)")  "*   Purpose: generate an event and evaluate expressions"
      write (u, "(A)")

      call syntax_model_file_init ()
      call syntax_pexpr_init ()

      write (u, "(A)")  "* Expression texts"
      write (u, "(A)")

      expr_text = "all Pt > 100 [s]"
      write (u, "(A,A)")  "selection = ", char (expr_text)
      call ifile_clear (ifile)
      call ifile_append (ifile, expr_text)
      call stream_init (stream, ifile)
      call parse_tree_init_lexpr (pt_selection, stream, .true.)
      call stream_final (stream)

      expr_text = "1 + sqrts_hat / sqrts"
      write (u, "(A,A)")  "reweight = ", char (expr_text)
      call ifile_clear (ifile)
```

2441

```
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_reweight, stream, .true.)
call stream_final (stream)

expr_text = "true"
write (u, "(A,A)")  "analysis = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_analysis, stream, .true.)
call stream_final (stream)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize test process event"

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process%set_var_list &
     (model_get_var_list_ptr (process%get_model_ptr ()))
call process_instance%setup_event_data ()

write (u, "(A)")
write (u, "(A)")  "* Initialize event object and set expressions"

allocate (event)
call event%basic_init ()

call event%set_selection (parse_tree_get_root_ptr (pt_selection))
call event%set_reweight (parse_tree_get_root_ptr (pt_reweight))
call event%set_analysis (parse_tree_get_root_ptr (pt_analysis))

call event%connect (process_instance, process%get_model_ptr ())
call var_list_append_real &
     (event%expr%var_list, var_str ("tolerance"), 0._default)
call event%setup_expressions ()

write (u, "(A)")
write (u, "(A)")  "* Generate test process event"

call process%generate_weighted_event (process_instance, 1)

write (u, "(A)")
write (u, "(A)")  "* Fill event object and evaluate expressions"
write (u, "(A)")

call event%generate (1, [0.4_default, 0.4_default])
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
```

```
        write (u, "(A)")  "* Cleanup"

        call event%final ()
        deallocate (event)

        call cleanup_test_process (process, process_instance)
        deallocate (process_instance)
        deallocate (process)

        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: events_3"

    end subroutine events_3
```

Recovering an event

Generate an event and store the particle set. Then reset the event record, recall the particle set, and recover the event from that.

⟨*Events: execute tests*⟩+≡
```
    call test (events_4, "events_4", &
        "recover event", &
        u, results)
```
⟨*Events: tests*⟩+≡
```
    subroutine events_4 (u)
      integer, intent(in) :: u
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(particle_set_t) :: particle_set
      type(model_list_t) :: model_list

      write (u, "(A)")  "* Test output: events_4"
      write (u, "(A)")  "*   Purpose: generate and recover an event"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Generate test process event and save particle set"
      write (u, "(A)")

      allocate (process)
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())
```

```
call event%generate (1, [0.4_default, 0.4_default])
call event%evaluate_expressions ()
call event%write (u)

particle_set = event%particle_set

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Recover event from particle set"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call event%select (1, 1, 1)
call event%set_particle_set_hard_proc (particle_set)
call event%recalculate (update_sqme = .true.)
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Transfer sqme and evaluate expressions"
write (u, "(A)")

call event%accept_sqme_prc ()
call event%accept_weight_prc ()
call event%check ()
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Reset contents"
write (u, "(A)")

call event%reset ()
event%transform_first%particle_set_exists = .false.
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (particle_set)
```

```
    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_4"

  end subroutine events_4
```

Partially Recovering an event

Generate an event and store the particle set. Then reset the event record, recall
the particle set, and recover the event as far as possible without recomputing
the squared matrix element.

⟨*Events: execute tests*⟩+≡
```
    call test (events_5, "events_5", &
         "partially recover event", &
         u, results)
```
⟨*Events: tests*⟩+≡
```
    subroutine events_5 (u)
      integer, intent(in) :: u
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(particle_set_t) :: particle_set
      real(default) :: sqme, weight
      type(model_list_t) :: model_list

      write (u, "(A)")  "* Test output: events_5"
      write (u, "(A)")  "*   Purpose: generate and recover an event"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Generate test process event and save particle set"
      write (u, "(A)")

      allocate (process)
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())
```

```
call event%generate (1, [0.4_default, 0.4_default])
call event%evaluate_expressions ()
call event%write (u)

particle_set = event%particle_set
sqme = event%sqme_ref
weight = event%weight_ref

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Recover event from particle set"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call event%select (1, 1, 1)
call event%set_particle_set_hard_proc (particle_set)
call event%recalculate (update_sqme = .false.)
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually set sqme and evaluate expressions"
write (u, "(A)")

call event%set (sqme_ref = sqme, weight_ref = weight)
call event%accept_sqme_ref ()
call event%accept_weight_ref ()
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set_final (particle_set)

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)
```

```
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: events_5"

    end subroutine events_5
```

Decays

Generate an event with subsequent decays.

⟨*Events: execute tests*⟩+≡
```
      call test (events_6, "events_6", &
           "decays", &
           u, results)
```

⟨*Events: tests*⟩+≡
```
    subroutine events_6 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(string_t) :: prefix, procname1, procname2
      type(process_library_t), target :: lib
      type(process_stack_t) :: process_stack
      class(evt_t), pointer :: evt_decay
      type(event_t), allocatable, target :: event
      type(process_t), pointer :: process
      type(process_instance_t), allocatable, target :: process_instance

      write (u, "(A)")  "* Test output: events_6"
      write (u, "(A)")  "*   Purpose: generate an event with subsequent decays"
      write (u, "(A)")

      write (u, "(A)")  "* Generate test process and decay"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)

      prefix = "events_6"
      procname1 = prefix // "_p"
      procname2 = prefix // "_d"
      call prepare_testbed &
           (lib, process_stack, model_list, model, prefix, os_data, &
           scattering=.true., decay=.true.)

      write (u, "(A)")  "* Initialize decay process"

      call model_set_unstable (model, 25, [procname2])

      process => process_stack%get_process_ptr (procname1)
```

2447

```
    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data ()
    call process_instance%init_simulation (1)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize event transform: decay"

    allocate (evt_decay_t :: evt_decay)
    call evt_decay%connect (process_instance, model, process_stack)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize event object"
    write (u, "(A)")

    allocate (event)
    call event%basic_init ()
    call event%connect (process_instance, model)
    call event%import_transform (evt_decay)

    call event%write (u, show_decay = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Generate event"
    write (u, "(A)")

    call event%generate (1, [0.4_default, 0.4_default])
    call event%evaluate_expressions ()
    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call event%final ()
    deallocate (event)

    call process_instance%final ()
    deallocate (process_instance)

    call process_stack%final ()

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_6"

  end subroutine events_6
```

Decays

Generate a decay event with varying options.

⟨*Events: execute tests*⟩+≡

```
      call test (events_7, "events_7", &
           "decay options", &
           u, results)
```

⟨*Events: tests*⟩+≡

```
     subroutine events_7 (u)
       integer, intent(in) :: u
       type(os_data_t) :: os_data
       type(model_list_t) :: model_list
       type(model_t), pointer :: model
       type(string_t) :: prefix, procname2
       type(process_library_t), target :: lib
       type(process_stack_t) :: process_stack
       type(process_t), pointer :: process
       type(process_instance_t), allocatable, target :: process_instance

       write (u, "(A)")  "* Test output: events_7"
       write (u, "(A)")  "*   Purpose: check decay options"
       write (u, "(A)")

       write (u, "(A)")  "* Prepare test process"
       write (u, "(A)")

       call syntax_model_file_init ()
       call os_data_init (os_data)

       prefix = "events_7"
       procname2 = prefix // "_d"
       call prepare_testbed &
            (lib, process_stack, model_list, model, prefix, os_data, &
            scattering=.false., decay=.true.)

       write (u, "(A)")  "* Generate decay event, default options"
       write (u, "(A)")

       call model_set_unstable (model, 25, [procname2])

       process => process_stack%get_process_ptr (procname2)
       allocate (process_instance)
       call process_instance%init (process)
       call process_instance%setup_event_data (model)
       call process_instance%init_simulation (1)

       call process%generate_weighted_event (process_instance, 1)
       call process_instance%write (u)

       call process_instance%final ()
       deallocate (process_instance)

       write (u, "(A)")
       write (u, "(A)")  "* Generate decay event, helicity-diagonal decay"
       write (u, "(A)")

       call model_set_unstable (model, 25, [procname2], diagonal = .true.)
```

2449

```
    process => process_stack%get_process_ptr (procname2)

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data (model)
    call process_instance%init_simulation (1)

    call process%generate_weighted_event (process_instance, 1)
    call process_instance%write (u)

    call process_instance%final ()
    deallocate (process_instance)

    write (u, "(A)")
    write (u, "(A)")  "* Generate decay event, isotropic decay, &
         &polarized final state"
    write (u, "(A)")

    call model_set_unstable (model, 25, [procname2], isotropic = .true.)
    call model_set_polarized (model, 6)
    call model_set_polarized (model, -6)

    process => process_stack%get_process_ptr (procname2)

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data (model)
    call process_instance%init_simulation (1)

    call process%generate_weighted_event (process_instance, 1)
    call process_instance%write (u)

    call process_instance%final ()
    deallocate (process_instance)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call process_stack%final ()

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_7"

end subroutine events_7
```

Chapter 18

Matrix Element Implementations

In this chapter, we define concrete implementations for the workflow, from matrix-element code generation via matrix-element access to integration and event generation.

18.1 Template matrix elements

Here, we provide template matrix elements that are in structure very similar to O'MEGA matrix elements, but do not need its infrastructure. Per default, the matrix elements are flat, i.e. the have the constant value one. Analogous to the O'MEGA implementation, this section implements the interface to the templates (via the makefile) and the driver.

⟨prc_template_me.f90⟩≡
  ⟨*File header*⟩

  module prc_template_me

    use iso_c_binding !NODEP!
    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use limits, only: TAB !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use os_interface
    use lorentz !NODEP!
    use flavors
    use sm_qcd
    use interactions
    use variables
    use models

    use process_constants
    use prclib_interfaces

```
    use prc_core_def
    use particle_specifiers
    use process_libraries
    use prc_core
```

⟨*Standard module head*⟩

⟨*Template matrix elements: public*⟩

⟨*Template matrix elements: types*⟩

⟨*Template matrix elements: interfaces*⟩

```
contains
```

⟨*Template matrix elements: procedures*⟩

⟨*Template matrix elements: tests*⟩

```
end module prc_template_me
```

18.1.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

⟨*Template matrix elements: public*⟩≡
```
    public :: template_me_def_t
```

⟨*Template matrix elements: types*⟩≡
```
    type, extends (prc_core_def_t) :: template_me_def_t
      contains
      ⟨Template matrix elements: template ME def: TBP⟩
    end type template_me_def_t
```

⟨*Template matrix elements: template ME def: TBP*⟩≡
```
    procedure, nopass :: type_string => template_me_def_type_string
```

⟨*Template matrix elements: procedures*⟩≡
```
    function template_me_def_type_string () result (string)
      type(string_t) :: string
      string = "template"
    end function template_me_def_type_string
```

Initialization: allocate the writer for the template matrix element. Also set any data for this process that the writer needs.

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
    procedure :: init => template_me_def_init
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_def_init &
        (object, model, prt_in, prt_out, unity)
      class(template_me_def_t), intent(out) :: object
      type(model_t), intent(in), target :: model
      type(string_t), dimension(:), intent(in) :: prt_in
```

```
      type(string_t), dimension(:), intent(in) :: prt_out
      logical, intent(in) :: unity
      allocate (template_me_writer_t :: object%writer)
      select type (writer => object%writer)
      type is (template_me_writer_t)
         call writer%init (model, prt_in, prt_out, unity)
      end select
    end subroutine template_me_def_init
```

Write/read process- and method-specific data.

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
    procedure :: write => template_me_def_write
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_def_write (object, unit)
      class(template_me_def_t), intent(in) :: object
      integer, intent(in) :: unit
      select type (writer => object%writer)
      type is (template_me_writer_t)
         call writer%write (unit)
      end select
    end subroutine template_me_def_write
```

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
    procedure :: read => template_me_def_read
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_def_read (object, unit)
      class(template_me_def_t), intent(out) :: object
      integer, intent(in) :: unit
      call msg_bug &
           ("WHIZARD template process definition: input not supported (yet)")
    end subroutine template_me_def_read
```

Allocate the driver for template matrix elements.

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
    procedure :: allocate_driver => template_me_def_allocate_driver
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_def_allocate_driver (object, driver, basename)
      class(template_me_def_t), intent(in) :: object
      class(prc_core_driver_t), intent(out), allocatable :: driver
      type(string_t), intent(in) :: basename
      allocate (template_me_driver_t :: driver)
    end subroutine template_me_def_allocate_driver
```

We need code:

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
    procedure, nopass :: needs_code => template_me_def_needs_code
```

⟨*Template matrix elements: procedures*⟩+≡
```
    function template_me_def_needs_code () result (flag)
      logical :: flag
      flag = .true.
    end function template_me_def_needs_code
```

These are the features that a template matrix element provides.
⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
    procedure, nopass :: get_features => template_me_def_get_features
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_def_get_features (features)
      type(string_t), dimension(:), allocatable, intent(out) :: features
      allocate (features (6))
      features = [ &
           var_str ("init"), &
           var_str ("update_alpha_s"), &
           var_str ("is_allowed"), &
           var_str ("new_event"), &
           var_str ("get_amplitude")]
    end subroutine template_me_def_get_features
```

The interface of the specific features.
⟨*Template matrix elements: interfaces*⟩≡
```
    abstract interface
       subroutine init_t (par) bind(C)
         import
         real(c_default_float), dimension(*), intent(in) :: par
       end subroutine init_t
    end interface

    abstract interface
       subroutine update_alpha_s_t (alpha_s) bind(C)
         import
         real(c_default_float), intent(in) :: alpha_s
       end subroutine update_alpha_s_t
    end interface

    abstract interface
       subroutine is_allowed_t (flv, hel, col, flag) bind(C)
         import
         integer(c_int), intent(in) :: flv, hel, col
         logical(c_bool), intent(out) :: flag
       end subroutine is_allowed_t
    end interface

    abstract interface
       subroutine new_event_t (p) bind(C)
         import
         real(c_default_float), dimension(0:3,*), intent(in) :: p
       end subroutine new_event_t
    end interface
```

```
      abstract interface
         subroutine get_amplitude_t (flv, hel, col, amp) bind(C)
            import
            integer(c_int), intent(in) :: flv, hel, col
            complex(c_default_complex), intent(out):: amp
         end subroutine get_amplitude_t
      end interface
```

Connect the template matrix element features with the process driver.

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
      procedure :: connect => template_me_def_connect
```

⟨*Template matrix elements: procedures*⟩+≡
```
      subroutine template_me_def_connect (def, lib_driver, i, proc_driver)
        class(template_me_def_t), intent(in) :: def
        class(prclib_driver_t), intent(in) :: lib_driver
        integer, intent(in) :: i
        class(prc_core_driver_t), intent(inout) :: proc_driver
        integer(c_int) :: pid, fid
        type(c_funptr) :: fptr
        select type (proc_driver)
        type is  (template_me_driver_t)
           pid = i
           fid = 1
           call lib_driver%get_fptr (pid, fid, fptr)
           call c_f_procpointer (fptr, proc_driver%init)
           fid = 2
           call lib_driver%get_fptr (pid, fid, fptr)
           call c_f_procpointer (fptr, proc_driver%update_alpha_s)
           fid = 3
           call lib_driver%get_fptr (pid, fid, fptr)
           call c_f_procpointer (fptr, proc_driver%is_allowed)
           fid = 4
           call lib_driver%get_fptr (pid, fid, fptr)
           call c_f_procpointer (fptr, proc_driver%new_event)
           fid = 5
           call lib_driver%get_fptr (pid, fid, fptr)
           call c_f_procpointer (fptr, proc_driver%get_amplitude)
        end select
      end subroutine template_me_def_connect
```

18.1.2 The Template Matrix element writer

Unlike O'MEGA, the template matrix element is directly written by the main
WHIZARD program, so there will be no entry in the makefile for calling an ex-
ternal program. The template matrix element writer is responsible for writing
interfaces and wrappers.

⟨*Template matrix elements: types*⟩+≡
```
      type, extends (prc_writer_f_module_t) :: template_me_writer_t
         type(model_t), pointer :: model => null ()
         type(string_t) :: model_name
         logical :: unity
```

2455

```
       type(string_t), dimension(:), allocatable :: prt_in
       type(string_t), dimension(:), allocatable :: prt_out
       integer :: n_in
       integer :: n_out
       integer :: n_tot
      contains
     ⟨Template matrix elements: template ME writer: TBP⟩
     end type template_me_writer_t
```

The reported type is the same as for the `template_me_def_t` type.

⟨*Template matrix elements: template ME writer: TBP*⟩≡
```
     procedure, nopass :: type_name => template_me_writer_type_name
```

⟨*Template matrix elements: procedures*⟩+≡
```
     function template_me_writer_type_name () result (string)
       type(string_t) :: string
       string = "template"
     end function template_me_writer_type_name
```

Taking into account the prefix for template ME module names.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
     procedure, nopass :: get_module_name => template_me_writer_get_module_name
```

⟨*Template matrix elements: procedures*⟩+≡
```
     function template_me_writer_get_module_name (id) result (name)
       type(string_t) :: name
       type(string_t), intent(in) :: id
       name = "tpr_" // id
     end function template_me_writer_get_module_name
```

Output. This is called by `template_me_def_write`.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
     procedure :: write => template_me_writer_write
```

⟨*Template matrix elements: procedures*⟩+≡
```
     subroutine template_me_writer_write (object, unit)
       class(template_me_writer_t), intent(in) :: object
       integer, intent(in) :: unit
       integer :: i, j
       write (unit, "(5x,A,I0)") "# incoming part. = ", object%n_in
       write (unit, "(7x,A)", advance="no") &
                             "     Initial state: "
       do i = 1, object%n_in - 1
          write (unit, "(1x,A)", advance="no") char (object%prt_in(i))
       end do
       write (unit, "(1x,A)") char (object%prt_in(object%n_in))
       write (unit, "(5x,A,I0)") "# outgoing part. = ", object%n_out
       write (unit, "(7x,A)", advance="no") &
                             "     Final state:   "
       do j = 1, object%n_out - 1
          write (unit, "(1x,A)", advance="no") char (object%prt_out(i))
       end do
       write (unit, "(1x,A)") char (object%prt_out(object%n_out))
       write (unit, "(5x,A,I0)") "# part. (total) = ", object%n_tot
```

```
    end subroutine template_me_writer_write
```

Initialize with process data.

⟨Template matrix elements: template ME writer: TBP⟩+≡
```
    procedure :: init => template_me_writer_init
```

⟨Template matrix elements: procedures⟩+≡
```
    subroutine template_me_writer_init (writer, model, &
         prt_in, prt_out, unity)
      class(template_me_writer_t), intent(out) :: writer
      type(model_t), intent(in), target :: model
      type(string_t), dimension(:), intent(in) :: prt_in
      type(string_t), dimension(:), intent(in) :: prt_out
      logical, intent(in) :: unity
      writer%model => model
      writer%model_name = model%get_name ()
      writer%n_in = size (prt_in)
      writer%n_out = size (prt_out)
      writer%n_tot = size (prt_in) + size (prt_out)
      allocate (writer%prt_in (size (prt_in)), source = prt_in)
      allocate (writer%prt_out (size (prt_out)), source = prt_out)
      writer%unity = unity
    end subroutine template_me_writer_init
```

The makefile is the driver file for the test matrix elements.

⟨Template matrix elements: template ME writer: TBP⟩+≡
```
    procedure :: write_makefile_code => template_me_write_makefile_code
```

⟨Template matrix elements: procedures⟩+≡
```
    subroutine template_me_write_makefile_code (writer, unit, id, os_data, testflag)
      class(template_me_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: testflag
      write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
      write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
      write (unit, "(5A)")  "clean-", char (id), ":"
      write (unit, "(5A)")  TAB, "rm -f tpr_", char (id), ".mod"
      write (unit, "(5A)")  TAB, "rm -f ", char (id), ".lo"
      write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
      write (unit, "(5A)")  "CLEAN_OBJECTS += tpr_", char (id), ".mod"
      write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
      write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
      write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
    end subroutine template_me_write_makefile_code
```

The source is written by this routine.

⟨Template matrix elements: template ME writer: TBP⟩+≡
```
    procedure :: write_source_code => template_me_write_source_code
```

⟨Template matrix elements: procedures⟩+≡
```
    subroutine template_me_write_source_code (writer, id)
      class(template_me_writer_t), intent(in) :: writer
```

```fortran
type(string_t), intent(in) :: id
integer, dimension(writer%n_in) :: prt_in, mult_in, col_in
type(flavor_t), dimension(1:writer%n_in) :: flv_in
integer, dimension(writer%n_out) :: prt_out, mult_out
integer, dimension(writer%n_tot) :: prt, mult
integer, dimension(:,:), allocatable :: sxxx
integer :: dummy, status
type(flavor_t), dimension(1:writer%n_out) :: flv_out
type(string_t) :: proc_str, comment_str
integer :: u, i, j
integer :: hel, hel_in, hel_out, fac, factor, col_fac
type(string_t) :: filename
comment_str = ""
do i = 1, writer%n_in
   comment_str = comment_str // writer%prt_in(i) // " "
end do
do j = 1, writer%n_out
   comment_str = comment_str // writer%prt_out(j) // " "
end do
do i = 1, writer%n_in
   prt_in(i) = model_get_particle_pdg (writer%model, writer%prt_in(i))
   call flavor_init (flv_in(i), prt_in(i), writer%model)
   mult_in(i) = flavor_get_multiplicity (flv_in(i))
   col_in(i) = abs(flavor_get_color_type (flv_in(i)))
   mult(i) = mult_in(i)
   end do
do j = 1, writer%n_out
   prt_out(j) = model_get_particle_pdg (writer%model, writer%prt_out(j))
   call flavor_init (flv_out(j), prt_out(j), writer%model)
   mult_out(j) = flavor_get_multiplicity (flv_out(j))
   mult(writer%n_in + j) = mult_out(j)
   end do
prt(1:writer%n_in) = prt_in(1:writer%n_in)
prt(writer%n_in+1:writer%n_tot) = prt_out(1:writer%n_out)
proc_str = converter (prt)
hel_in = product (mult_in)
hel_out = product (mult_out)
col_fac = product (col_in)
hel = hel_in * hel_out
fac = hel
dummy = 1
factor = 1
if (writer%n_out >= 3) then
   do i = 3, writer%n_out
      factor = factor * (i - 2) * (i - 1)
   end do
end if
factor = factor * col_fac
allocate (sxxx(1:hel,1:writer%n_tot))
call create_spin_table (dummy,hel,fac,mult,sxxx)
call msg_message ("Writing test matrix element for process '" &
     // char (id) // "'")
filename = id // ".f90"
u = free_unit ()
```

```
open (unit=u, file=char(filename), action="write")
write (u, "(A)") "! File generated automatically by WHIZARD"
write (u, "(A)") "!                                           "
write (u, "(A)") "! Note that irresp. of what you demanded WHIZARD"
write (u, "(A)") "! treats this as colorless process          "
write (u, "(A)") "!                                           "
write (u, "(A)") "module tpr_" // char(id)
write (u, "(A)") "                                            "
write (u, "(A)") "  use kinds"
write (u, "(A)") "  use omega_color, OCF => omega_color_factor"
write (u, "(A)") "                                            "
write (u, "(A)") "  implicit none"
write (u, "(A)") "  private"
write (u, "(A)") "                                            "
write (u, "(A)") "  public :: md5sum"
write (u, "(A)") "  public :: number_particles_in, number_particles_out"
write (u, "(A)") "  public :: number_spin_states, spin_states"
write (u, "(A)") "  public :: number_flavor_states, flavor_states"
write (u, "(A)") "  public :: number_color_flows, color_flows"
write (u, "(A)") "  public :: number_color_indices, number_color_factors, &"
write (u, "(A)") "     color_factors, color_sum, openmp_supported"
write (u, "(A)") "  public :: init, final, update_alpha_s"
write (u, "(A)") "                                            "
write (u, "(A)") "  public :: new_event, is_allowed, get_amplitude"
write (u, "(A)") "      "
write (u, "(A)") "  real(default), parameter :: &"
write (u, "(A)") "       & conv = 0.38937966e12_default"
write (u, "(A)") "      "
write (u, "(A)") "  real(default), parameter :: &"
write (u, "(A)") "       & pi = 3.1415926535897932384626433832795028841972_default"
write (u, "(A)") "      "
write (u, "(A)") "  real(default), parameter :: &"
if (writer%unity) then
   write (u, "(A)") "                 & const = 1"
else
   write (u, "(A,1x,I0,A)") "      & const = (16 * pi / conv) * " &
        // "(16 * pi**2)**(", writer%n_out, "-2) "
end if
write (u, "(A)") "       "
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_prt = ", &
   writer%n_tot
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_in = ", &
   writer%n_in
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_out = ", &
   writer%n_out
write (u, "(A)") "  integer, parameter, private :: n_cflow = 1"
write (u, "(A)") "  integer, parameter, private :: n_cindex = 2"
write (u, "(A)") "  !!! We ignore tensor products and take only one flavor state."
write (u, "(A)") "  integer, parameter, private :: n_flv = 1"
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_hel = ", hel
write (u, "(A)") "                                            "
write (u, "(A)") "  logical, parameter, private :: T = .true."
write (u, "(A)") "  logical, parameter, private :: F = .false."
write (u, "(A)") "                                            "
```

```
         do i = 1, hel
            write (u, "(A)") "  integer, dimension(n_prt), parameter, private :: &"
            write (u, "(A)") "     " // s_conv(i) // " = (/ " // &
                 char(converter(sxxx(i,1:writer%n_tot))) // " /)"
         end do
         write (u, "(A)") "  integer, dimension(n_prt,n_hel), parameter, private :: table_spin_states =
         write (u, "(A)") "    reshape ( (/ & "
         do i = 1, hel-1
            write (u, "(A)") "                  " // s_conv(i) // ", & "
         end do
         write (u, "(A)") "                  " // s_conv(hel) // " & "
         write (u, "(A)") "              /), (/ n_prt, n_hel /) )"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  integer, dimension(n_prt), parameter, private :: &"
         write (u, "(A)") "    f0001 = (/ " // char(proc_str) // " /)   !  " // char(comment_str)
         write (u, "(A)") "  integer, dimension(n_prt,n_flv), parameter, private :: table_flavor_states
         write (u, "(A)") "    reshape ( (/ f0001 /), (/ n_prt, n_flv /) )"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  integer, dimension(n_cindex, n_prt), parameter, private :: &"
         write (u, "(A)") "    c0001 = reshape ( (/ " // char (dummy_colorizer (flv_in)) // " " // &
           (repeat ("0,0, ", writer%n_out-1)) // "0,0 /), " // " (/ n_cindex, n_prt /) )"
         write (u, "(A)") "  integer, dimension(n_cindex, n_prt, n_cflow), parameter, private :: &"
         write (u, "(A)") "    table_color_flows = reshape ( (/ c0001 /), (/ n_cindex, n_prt, n_cflow /)
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  logical, dimension(n_prt), parameter, private :: & "
         write (u, "(A)") "    g0001 = (/ "  // (repeat ("F, ", writer%n_tot-1)) // "F /) "
         write (u, "(A)") "  logical, dimension(n_prt, n_cflow), parameter, private " &
              // ":: table_ghost_flags = &"
         write (u, "(A)") "    reshape ( (/ g0001 /), (/ n_prt, n_cflow /) )"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  integer, parameter, private :: n_cfactors = 1"
         write (u, "(A)") "  type(OCF), dimension(n_cfactors), parameter, private :: &"
         write (u, "(A)") "    table_color_factors = (/  OCF(1,1,+1._default) /)"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  logical, dimension(n_flv), parameter, private :: a0001 = (/ T /)"
         write (u, "(A)") "  logical, dimension(n_flv, n_cflow), parameter, private :: &"
         write (u, "(A)") "    flv_col_is_allowed = reshape ( (/ a0001 /), (/ n_flv, n_cflow /) )"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  complex(default), dimension (n_flv, n_hel, n_cflow), private, save :: amp"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  logical, dimension(n_hel), private, save :: hel_is_allowed = T"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "contains"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  pure function md5sum ()"
         write (u, "(A)") "    character(len=32) :: md5sum"
         write (u, "(A)") "    ! DON'T EVEN THINK of modifying the following line!"
         write (u, "(A)") "    md5sum = """ // writer%md5sum // """"
         write (u, "(A)") "  end function md5sum"
         write (u, "(A)") "                                                     "
         write (u, "(A)") "  subroutine init (par)"
         write (u, "(A)") "    real(default), dimension(*), intent(in) :: par"
         write (u, "(A)") "  end subroutine init"
         write (u, "(A)") "                                                     "
```

```
write (u, "(A)") "  subroutine final ()"
write (u, "(A)") "  end subroutine final"
write (u, "(A)") "                                                    "
write (u, "(A)") "  subroutine update_alpha_s (alpha_s)"
write (u, "(A)") "    real(default), intent(in) :: alpha_s"
write (u, "(A)") "  end subroutine update_alpha_s"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function number_particles_in () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = n_in"
write (u, "(A)") "  end function number_particles_in"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function number_particles_out () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = n_out"
write (u, "(A)") "  end function number_particles_out"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function number_spin_states () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_spin_states, dim=2)"
write (u, "(A)") "  end function number_spin_states"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure subroutine spin_states (a)"
write (u, "(A)") "    integer, dimension(:,:), intent(out) :: a"
write (u, "(A)") "    a = table_spin_states"
write (u, "(A)") "  end subroutine spin_states"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function number_flavor_states () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = 1"
write (u, "(A)") "  end function number_flavor_states"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure subroutine flavor_states (a)"
write (u, "(A)") "    integer, dimension(:,:), intent(out) :: a"
write (u, "(A)") "    a = table_flavor_states"
write (u, "(A)") "  end subroutine flavor_states"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function number_color_indices () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size(table_color_flows, dim=1)"
write (u, "(A)") "  end function number_color_indices"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure subroutine color_factors (cf)"
write (u, "(A)") "    type(OCF), dimension(:), intent(out) :: cf"
write (u, "(A)") "    cf = table_color_factors"
write (u, "(A)") "  end subroutine color_factors"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function color_sum (flv, hel) result (amp2)"
write (u, "(A)") "    integer, intent(in) :: flv, hel"
write (u, "(A)") "    real(kind=default) :: amp2"
write (u, "(A)") "    amp2 = real (omega_color_sum (flv, hel, amp, table_color_factors))"
write (u, "(A)") "  end function color_sum"
write (u, "(A)") "                                                    "
write (u, "(A)") "  pure function number_color_flows () result (n)"
```

```
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_color_flows, dim=3)"
write (u, "(A)") "  end function number_color_flows"
write (u, "(A)") "                                                  "
write (u, "(A)") "  pure subroutine color_flows (a, g)"
write (u, "(A)") "    integer, dimension(:,:,:), intent(out) :: a"
write (u, "(A)") "    logical, dimension(:,:), intent(out) :: g"
write (u, "(A)") "    a = table_color_flows"
write (u, "(A)") "    g = table_ghost_flags"
write (u, "(A)") "  end subroutine color_flows"
write (u, "(A)") "                                                  "
write (u, "(A)") "  pure function number_color_factors () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_color_factors)"
write (u, "(A)") "  end function number_color_factors"
write (u, "(A)") "                                                  "
write (u, "(A)") "  pure function openmp_supported () result (status)"
write (u, "(A)") "    logical :: status"
write (u, "(A)") "    status = .false."
write (u, "(A)") "  end function openmp_supported"
write (u, "(A)") "                                                  "
write (u, "(A)") "  subroutine new_event (p)"
write (u, "(A)") "    real(default), dimension(0:3,*), intent(in) :: p"
write (u, "(A)") "    call calculate_amplitudes (amp, p)"
write (u, "(A)") "  end subroutine new_event"
write (u, "(A)") "                                                  "
write (u, "(A)") "  pure function is_allowed (flv, hel, col) result (yorn)"
write (u, "(A)") "    logical :: yorn"
write (u, "(A)") "    integer, intent(in) :: flv, hel, col"
write (u, "(A)") "    yorn = hel_is_allowed(hel) .and. flv_col_is_allowed(flv,col)"
write (u, "(A)") "  end function is_allowed"
write (u, "(A)") "                                                  "
write (u, "(A)") "  pure function get_amplitude (flv, hel, col) result (amp_result)"
write (u, "(A)") "    complex(default) :: amp_result"
write (u, "(A)") "    integer, intent(in) :: flv, hel, col"
write (u, "(A)") "    amp_result = amp (flv, hel, col)"
write (u, "(A)") "  end function get_amplitude"
write (u, "(A)") "                                                  "
write (u, "(A)") "  pure subroutine calculate_amplitudes (amp, k)"
write (u, "(A)") "    complex(default), dimension(:,:,:), intent(out) :: amp"
write (u, "(A)") "    real(default), dimension(0:3,*), intent(in) :: k"
write (u, "(A)") "    real(default) :: fac"
write (u, "(A)") "    integer :: i"
write (u, "(A)") "    ! We give all helicities the same weight!"
if (writer%unity) then
   write (u, "(A,1x,I0,1x,A)") "    fac = ", col_fac
   write (u, "(A)") "    amp = const * sqrt(fac)"
else
   write (u, "(A,1x,I0,1x,A)") "    fac = ", factor
   write (u, "(A)") "    amp = sqrt((2 * (k(0,1)*k(0,2) &"
   write (u, "(A,1x,I0,A)") "             - dot_product (k(1:,1), k(1:,2)))) ** (3-", &
                         writer%n_out, ")) * sqrt(const * fac)"
end if
write (u, "(A,1x,I0,A)") "    amp = amp / sqrt(", hel_out, "._default)"
```

2462

```
      write (u, "(A)") "  end subroutine calculate_amplitudes"
      write (u, "(A)") "                                                  "
      write (u, "(A)") "end module tpr_" // char(id)
      close (u, iostat=status)
      deallocate (sxxx)
contains
    function s_conv (num) result (chrt)
      integer, intent(in) :: num
      character(len=10) :: chrt
      write (chrt, "(I10)") num
      chrt = trim(adjustl(chrt))
      if (num < 10) then
         chrt = "s000" // chrt
      else if (num < 100) then
         chrt = "s00" // chrt
      else if (num < 1000) then
         chrt = "s0" // chrt
      else
         chrt = "s" // chrt
      end if
    end function s_conv
    function converter (flv) result (str)
      integer, dimension(:), intent(in) :: flv
      type(string_t) :: str
      character(len=150), dimension(size(flv)) :: chrt
      integer :: i
      str = ""
      do i = 1, size(flv) - 1
         write (chrt(i), "(I10)") flv(i)
         str = str // var_str(trim(adjustl(chrt(i)))) // ", "
      end do
      write (chrt(size(flv)), "(I10)") flv(size(flv))
      str = str // trim(adjustl(chrt(size(flv))))
    end function converter
    integer function sj (j,m)
      integer, intent(in) :: j, m
      if (((j == 1) .and. (m == 1)) .or. &
          ((j == 2) .and. (m == 2)) .or. &
          ((j == 3) .and. (m == 3)) .or. &
          ((j == 4) .and. (m == 3)) .or. &
          ((j == 5) .and. (m == 4))) then
         sj = 1
      else if (((j == 2) .and. (m == 1)) .or. &
          ((j == 3) .and. (m == 1)) .or. &
          ((j == 4) .and. (m == 2)) .or. &
          ((j == 5) .and. (m == 2))) then
         sj = -1
      else if (((j == 3) .and. (m == 2)) .or. &
          ((j == 5) .and. (m == 3))) then
         sj = 0
      else if (((j == 4) .and. (m == 1)) .or. &
          ((j == 5) .and. (m == 1))) then
         sj = -2
      else if (((j == 4) .and. (m == 4)) .or. &
```

2463

```
      ((j == 5) .and. (m == 5))) then
         sj = 2
      else
         call msg_fatal ("template_me_write_source_code: Wrong spin type")
      end if
    end function sj
    recursive subroutine create_spin_table (index, nhel, fac, mult, inta)
      integer, intent(inout) :: index, fac
      integer, intent(in) :: nhel
      integer, dimension(:), intent(in) :: mult
      integer, dimension(nhel,size(mult)), intent(out) :: inta
      integer :: j
      if (index > size(mult)) return
      fac = fac / mult(index)
      do j = 1, nhel
         inta(j,index) = sj (mult(index),mod(((j-1)/fac),mult(index))+1)
      end do
      index = index + 1
      call create_spin_table (index, nhel, fac, mult, inta)
    end subroutine create_spin_table
    function dummy_colorizer (flv) result (str)
      type(flavor_t), dimension(:), intent(in) :: flv
      type(string_t) :: str
      integer :: i, k
      str = ""
      k = 0
      do i = 1, size(flv)
         k = k + 1
         select case (flavor_get_color_type (flv(i)))
         case (1,-1)
            str = str // "0,0, "
         case (3)
            str = str // int2string(k) // ",0, "
         case (-3)
            str = str // "0," // int2string(-k) // ", "
         case (8)
            str = str // int2string(k) // "," // int2string(-k-1) // ", "
            k = k + 1
         case default
            call msg_error ("Color type not supported.")
         end select
      end do
      str = adjustl(trim(str))
    end function dummy_colorizer
  end subroutine template_me_write_source_code
```

Return the name of a procedure that implements a given feature, as it is provided by the template matrix-element code. Template ME names are chosen completely in analogy to the O'MEGA matrix element conventions.

⟨Template matrix elements: template ME writer: TBP⟩+≡
```
    procedure, nopass :: get_procname => template_me_writer_get_procname
```

⟨Template matrix elements: procedures⟩+≡
```
    function template_me_writer_get_procname (feature) result (name)
```

```
      type(string_t) :: name
      type(string_t), intent(in) :: feature
      select case (char (feature))
      case ("n_in");    name = "number_particles_in"
      case ("n_out");   name = "number_particles_out"
      case ("n_flv");   name = "number_flavor_states"
      case ("n_hel");   name = "number_spin_states"
      case ("n_col");   name = "number_color_flows"
      case ("n_cin");   name = "number_color_indices"
      case ("n_cf");    name = "number_color_factors"
      case ("flv_state");  name = "flavor_states"
      case ("hel_state");  name = "spin_states"
      case ("col_state");  name = "color_flows"
      case default
         name = feature
      end select
    end function template_me_writer_get_procname
```

The interfaces for the template-specific features.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
    procedure :: write_interface => template_me_write_interface
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_write_interface (writer, unit, id, feature)
      class(template_me_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(string_t), intent(in) :: feature
      type(string_t) :: name
      name = writer%get_c_procname (id, feature)
      write (unit, "(2x,9A)")  "interface"
      select case (char (feature))
      case ("init")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " (par) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), dimension(*), &
               &intent(in) :: par"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("update_alpha_s")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: alpha_s"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("is_allowed")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " &
               &(flv, hel, col, flag) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
         write (unit, "(7x,9A)")  "logical(c_bool), intent(out) :: flag"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("new_event")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " (p) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), dimension(0:3,*), &
```

```
            &intent(in) :: p"
      write (unit, "(5x,9A)")  "end subroutine ", char (name)
   case ("get_amplitude")
      write (unit, "(5x,9A)")  "subroutine ", char (name), " &
            &(flv, hel, col, amp) bind(C)"
      write (unit, "(7x,9A)")  "import"
      write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(7x,9A)")  "complex(c_default_complex), intent(out) &
            &:: amp"
      write (unit, "(5x,9A)")  "end subroutine ", char (name)
   end select
   write (unit, "(2x,9A)")  "end interface"
end subroutine template_me_write_interface
```

The wrappers have to take into account conversion between C and Fortran data types.

NOTE: The case c_default_float ≠ default is not yet covered.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
   procedure :: write_wrapper => template_me_write_wrapper
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_write_wrapper (writer, unit, id, feature)
  class(template_me_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  type(string_t) :: name
  name = writer%get_c_procname (id, feature)
  write (unit, *)
  select case (char (feature))
  case ("init")
     write (unit, "(9A)")  "subroutine ", char (name), " (par) bind(C)"
     write (unit, "(2x,9A)")  "use iso_c_binding"
     write (unit, "(2x,9A)")  "use kinds"
     write (unit, "(2x,9A)")  "use tpr_", char (id)
     write (unit, "(2x,9A)")  "real(c_default_float), dimension(*), &
           &intent(in) :: par"
     if (c_default_float == default) then
        write (unit, "(2x,9A)")  "call ", char (feature), " (par)"
     end if
     write (unit, "(9A)")  "end subroutine ", char (name)
  case ("update_alpha_s")
     write (unit, "(9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
     write (unit, "(2x,9A)")  "use iso_c_binding"
     write (unit, "(2x,9A)")  "use kinds"
     write (unit, "(2x,9A)")  "use tpr_", char (id)
     if (c_default_float == default) then
        write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
              &:: alpha_s"
        write (unit, "(2x,9A)")  "call ", char (feature), " (alpha_s)"
     end if
     write (unit, "(9A)")  "end subroutine ", char (name)
  case ("is_allowed")
     write (unit, "(9A)")  "subroutine ", char (name), &
           " (flv, hel, col, flag) bind(C)"
```

```
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use tpr_", char (id)
        write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
        write (unit, "(2x,9A)")  "logical(c_bool), intent(out) :: flag"
        write (unit, "(2x,9A)")  "flag = ", char (feature), &
              " (int (flv), int (hel), int (col))"
        write (unit, "(9A)")  "end subroutine ", char (name)
     case ("new_event")
        write (unit, "(9A)")  "subroutine ", char (name), " (p) bind(C)"
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use tpr_", char (id)
        if (c_default_float == default) then
           write (unit, "(2x,9A)")  "real(c_default_float), dimension(0:3,*), &
                 &intent(in) :: p"
           write (unit, "(2x,9A)")  "call ", char (feature), " (p)"
        end if
        write (unit, "(9A)")  "end subroutine ", char (name)
     case ("get_amplitude")
        write (unit, "(9A)")  "subroutine ", char (name), &
              " (flv, hel, col, amp) bind(C)"
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use tpr_", char (id)
        write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
        write (unit, "(2x,9A)")  "complex(c_default_complex), intent(out) &
              &:: amp"
        write (unit, "(2x,9A)")  "amp = ", char (feature), &
              " (int (flv), int (hel), int (col))"
        write (unit, "(9A)")  "end subroutine ", char (name)
     end select
   end subroutine template_me_write_wrapper
```

18.1.3 Driver

⟨*Template matrix elements: types*⟩+≡

```
   type, extends (prc_core_driver_t) :: template_me_driver_t
     procedure(init_t), nopass, pointer :: &
          init => null ()
     procedure(update_alpha_s_t), nopass, pointer :: &
          update_alpha_s => null ()
     procedure(is_allowed_t), nopass, pointer :: &
          is_allowed => null ()
     procedure(new_event_t), nopass, pointer :: &
          new_event => null ()
     procedure(get_amplitude_t), nopass, pointer :: &
          get_amplitude => null ()
    contains
    ⟨Template matrix elements: template ME driver: TBP⟩
   end type template_me_driver_t
```

The reported type is the same as for the `template_me_def_t` type.

⟨*Template matrix elements: template ME driver: TBP*⟩≡
```
procedure, nopass :: type_name => template_me_driver_type_name
```

⟨*Template matrix elements: procedures*⟩+≡
```
function template_me_driver_type_name () result (string)
  type(string_t) :: string
  string = "template"
end function template_me_driver_type_name
```

18.1.4 High-level process definition

This procedure wraps the details filling a process-component definition entry as appropriate for an template matrix element.

  NOTE: For calling the `import_component` method, we must explicitly address the `process_def_t` parent object. The natural way to call the method of the extended type triggers a bug in gfortran 4.6. The string array arguments `prt_in` and `prt_out` become corrupted and cause a segfault.

⟨*Template matrix elements: public*⟩+≡
```
public :: template_me_make_process_component
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_make_process_component (entry, component_index, &
      model, model_name, prt_in, prt_out, unity)
  class(process_def_entry_t), intent(inout) :: entry
  integer, intent(in) :: component_index
  type(string_t), intent(in) :: model_name
  type(model_t), intent(in), target :: model
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  logical, intent(in) :: unity
  class(prc_core_def_t), allocatable :: def
  allocate (template_me_def_t :: def)
  select type (def)
  type is (template_me_def_t)
     call def%init (model, prt_in, prt_out, unity)
  end select
  call entry%process_def_t%import_component (component_index, &
      n_out = size (prt_out), &
      prt_in  = new_prt_spec (prt_in), &
      prt_out = new_prt_spec (prt_out), &
      method = var_str ("template"), &
      variant = def)
end subroutine template_me_make_process_component
```

18.1.5 The `prc_template_me_t` wrapper

This is an instance of the generic `prc_core_t` object. It contains a pointer to the process definition (`template_me_def_t`), a data component (`process_constants_t`), and the matrix-element driver (`template_me_driver_t`).

⟨*Template matrix elements: public*⟩+≡

```
        public :: prc_template_me_t
```

⟨*Template matrix elements: types*⟩+≡
```
    type, extends (prc_core_t) :: prc_template_me_t
      real(default), dimension(:), allocatable :: par
     contains
     ⟨Template matrix elements: prc template ME: TBP⟩
    end type prc_template_me_t
```

The workspace associated to a `prc_template_me_t` object contains a single flag. The flag is used to suppress re-evaluating the matrix element for each quantum-number combination, after the first amplitude belonging to a given kinematics has been computed.

We can also store the value of a running coupling once it has been calculated for an event. The default value is negative, which indicates an undefined value in this context.

⟨*Template matrix elements: types*⟩+≡
```
    type, extends (workspace_t) :: template_me_state_t
      logical :: new_kinematics = .true.
      real(default) :: alpha_qcd = -1
     contains
      procedure :: write => template_me_state_write
    end type template_me_state_t
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine template_me_state_write (object, unit)
      class(template_me_state_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A,L1)")  "Template ME state: new kinematics = ", &
          object%new_kinematics
    end subroutine template_me_state_write
```

Allocate the workspace with the above specific type.

⟨*Template matrix elements: prc template ME: TBP*⟩≡
```
    procedure :: allocate_workspace => prc_template_me_allocate_workspace
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine prc_template_me_allocate_workspace (object, tmp)
      class(prc_template_me_t), intent(in) :: object
      class(workspace_t), intent(inout), allocatable :: tmp
      allocate (template_me_state_t :: tmp)
    end subroutine prc_template_me_allocate_workspace
```

The following procedures are inherited from the base type as deferred, thus must be implemented. The corresponding unit tests are skipped here; the procedures are tested when called from the `processes` module.

Output: print just the ID of the associated matrix element. Then display any stored parameters.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
    procedure :: write => prc_template_me_write
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine prc_template_me_write (object, unit)
      class(prc_template_me_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(3x,A)", advance="no")  "Template process core:"
      if (object%data_known) then
         write (u, "(1x,A)")  char (object%data%id)
      else
         write (u, "(1x,A)")  "[undefined]"
      end if
      if (allocated (object%par)) then
         write (u, "(3x,A)")  "Parameter array:"
         do i = 1, size (object%par)
            write (u, "(5x,I0,1x,ES17.10)")  i, object%par(i)
         end do
      end if
    end subroutine prc_template_me_write
```

Temporarily store the parameter array inside the `prc_template_me` object, so
we can use it later during the actual initialization.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
    procedure :: set_parameters => prc_template_me_set_parameters
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine prc_template_me_set_parameters (prc_template_me, model)
      class(prc_template_me_t), intent(inout) :: prc_template_me
      type(model_t), intent(in), target, optional :: model
      if (present (model)) then
         if (allocated (prc_template_me%par))  deallocate (prc_template_me%par)
         call model_parameters_to_c_array (model, prc_template_me%par)
      end if
    end subroutine prc_template_me_set_parameters
```

To fully initialize the process core, we perform base initialization, then initialize
the external matrix element code.

This procedure overrides the `init` method of the base type, which we never-
theless can access via its binding `base_init`. When done, we have an allocated
driver. The driver will call the `init` procedure for the external matrix element,
and thus transfer the parameter set to where it finally belongs.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
    procedure :: init => prc_template_me_init
```

⟨*Template matrix elements: procedures*⟩+≡
```
    subroutine prc_template_me_init (object, def, lib, id, i_component)
      class(prc_template_me_t), intent(inout) :: object
      class(prc_core_def_t), intent(in), target :: def
      type(process_library_t), intent(in), target :: lib
      type(string_t), intent(in) :: id
      integer, intent(in) :: i_component
      call object%base_init (def, lib, id, i_component)
      call object%activate_parameters ()
```

```
      end subroutine prc_template_me_init
```

Activate the stored parameters by transferring them to the external matrix element.

⟨Template matrix elements: prc template ME: TBP⟩+≡
```
    procedure :: activate_parameters => prc_template_me_activate_parameters
```

⟨Template matrix elements: procedures⟩+≡
```
    subroutine prc_template_me_activate_parameters (object)
      class (prc_template_me_t), intent(inout) :: object
      if (allocated (object%driver)) then
         if (allocated (object%par)) then
            select type (driver => object%driver)
            type is (template_me_driver_t)
               if (associated (driver%init))  call driver%init (object%par)
            end select
         else
            call msg_bug ("prc_template_me_activate: parameter set is not allocated")
         end if
      else
         call msg_bug ("prc_template_me_activate: driver is not allocated")
      end if
    end subroutine prc_template_me_activate_parameters
```

The template ME process is an independent process which needs its own Monte Carlo parameter set for integration.

⟨Template matrix elements: prc template ME: TBP⟩+≡
```
    procedure :: needs_mcset => prc_template_me_needs_mcset
```

⟨Template matrix elements: procedures⟩+≡
```
    function prc_template_me_needs_mcset (object) result (flag)
      class(prc_template_me_t), intent(in) :: object
      logical :: flag
      flag = .true.
    end function prc_template_me_needs_mcset
```

There is only one term for this tree-level process.

⟨Template matrix elements: prc template ME: TBP⟩+≡
```
    procedure :: get_n_terms => prc_template_me_get_n_terms
```

⟨Template matrix elements: procedures⟩+≡
```
    function prc_template_me_get_n_terms (object) result (n)
      class(prc_template_me_t), intent(in) :: object
      integer :: n
      n = 1
    end function prc_template_me_get_n_terms
```

Tell whether a particular combination of flavor, helicity, color is allowed. Here we have to consult the matrix-element driver.

⟨Template matrix elements: prc template ME: TBP⟩+≡
```
    procedure :: is_allowed => prc_template_me_is_allowed
```

⟨*Template matrix elements: procedures*⟩+≡
```
   function prc_template_me_is_allowed (object, i_term, f, h, c) result (flag)
      class(prc_template_me_t), intent(in) :: object
      integer, intent(in) :: i_term, f, h, c
      logical :: flag
      logical(c_bool) :: cflag
      select type (driver => object%driver)
      type is (template_me_driver_t)
         call driver%is_allowed (f, h, c, cflag)
         flag = cflag
      end select
   end function prc_template_me_is_allowed
```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

We reset the `new_kinematics` flag, so that the next call to `compute_amplitude` will evaluate the matrix element.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
     procedure :: compute_hard_kinematics => &
         prc_template_me_compute_hard_kinematics
```

⟨*Template matrix elements: procedures*⟩+≡
```
     subroutine prc_template_me_compute_hard_kinematics &
         (object, p_seed, i_term, int_hard, tmp)
      class(prc_template_me_t), intent(in) :: object
      type(vector4_t), dimension(:), intent(in) :: p_seed
      integer, intent(in) :: i_term
      type(interaction_t), intent(inout) :: int_hard
      class(workspace_t), intent(inout), allocatable :: tmp
      call interaction_set_momenta (int_hard, p_seed)
      if (allocated (tmp)) then
         select type (tmp)
         type is (template_me_state_t);  tmp%new_kinematics = .true.
         end select
      end if
     end subroutine prc_template_me_compute_hard_kinematics
```

This procedure is not called for `prc_template_me_t`, just a placeholder.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
     procedure :: compute_eff_kinematics => &
         prc_template_me_compute_eff_kinematics
```

⟨*Template matrix elements: procedures*⟩+≡
```
     subroutine prc_template_me_compute_eff_kinematics &
         (object, i_term, int_hard, int_eff, tmp)
      class(prc_template_me_t), intent(in) :: object
      integer, intent(in) :: i_term
      type(interaction_t), intent(in) :: int_hard
      type(interaction_t), intent(inout) :: int_eff
      class(workspace_t), intent(inout), allocatable :: tmp
     end subroutine prc_template_me_compute_eff_kinematics
```

Recover the momenta within the hard interaction. For tree-level processes, this is a trivial copy of the incoming seed and outgoing effective momenta. The effective interaction is a pointer to the hard interaction, so the latter is ignored.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
  procedure :: recover_kinematics => prc_template_me_recover_kinematics
```

⟨*Template matrix elements: tests*⟩≡
```
  subroutine prc_template_me_recover_kinematics &
      (object, p_seed, int_hard, int_eff, tmp)
    class(prc_template_me_t), intent(in) :: object
    type(vector4_t), dimension(:), intent(inout) :: p_seed
    type(interaction_t), intent(inout) :: int_hard
    type(interaction_t), intent(inout) :: int_eff
    class(workspace_t), intent(inout), allocatable :: tmp
    integer :: n_in
    n_in = interaction_get_n_in (int_eff)
    call interaction_set_momenta (int_eff, p_seed(1:n_in), outgoing = .false.)
    p_seed(n_in+1:) = interaction_get_momenta (int_eff, outgoing = .true.)
  end subroutine prc_template_me_recover_kinematics
```

Compute the amplitude. For the tree-level process, we can ignore the scale settings. The term index `j` is also irrelevant.

We first call `new_event` for the given momenta (which we must unpack), then retrieve the amplitude value for the given quantum numbers.

If the `tmp` status flag is present, we can make sure that we call `new_event` only once for a given kinematics. After the first call, we unset the `new_kinematics` flag.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
  procedure :: compute_amplitude => prc_template_me_compute_amplitude
```

⟨*Template matrix elements: procedures*⟩+≡
```
  function prc_template_me_compute_amplitude &
      (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
    class(prc_template_me_t), intent(in) :: object
    integer, intent(in) :: j
    type(vector4_t), dimension(:), intent(in) :: p
    integer, intent(in) :: f, h, c
    real(default), intent(in) :: fac_scale, ren_scale
    class(workspace_t), intent(inout), allocatable, optional :: tmp
    complex(default) :: amp
    integer :: n_tot, i
    real(c_default_float), dimension(:,:), allocatable :: parray
    complex(c_default_complex) :: camp
    logical :: new_event
    select type (driver => object%driver)
    type is (template_me_driver_t)
      new_event = .true.
      if (present (tmp)) then
        if (allocated (tmp)) then
          select type (tmp)
          type is (template_me_state_t)
            new_event = tmp%new_kinematics
            tmp%new_kinematics = .false.
          end select
```

```
            end if
        end if
        if (new_event) then
            n_tot = object%data%n_in + object%data%n_out
            allocate (parray (0:3, n_tot))
            forall (i = 1:n_tot)  parray(:,i) = vector4_get_components (p(i))
            call driver%new_event (parray)
        end if
        if (object%is_allowed (1, f, h, c)) then
            call driver%get_amplitude &
                 (int (f, c_int), int (h, c_int), int (c, c_int), camp)
            amp = camp
        else
            amp = 0
        end if
    end select
  end function prc_template_me_compute_amplitude
```

We do not overwrite the `prc_core_t` routine for α_s.

18.1.6 Test

This is the master for calling self-test procedures. There is a separate test for
testing O'MEGA diagram generation as this depends on a working analysis setup.

⟨*Template matrix elements: public*⟩+≡
```
    public :: prc_template_me_test
```

⟨*Template matrix elements: tests*⟩+≡
```
    subroutine prc_template_me_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Template matrix elements: execute tests⟩
  end subroutine prc_template_me_test
```

Generate, compile and load a simple process matrix element

The process is $e^+e^- \to \mu^+\mu^-$ for vanishing masses and $e = 0.3$. We initialize
the process, build the library, and compute a particular matrix element for
momenta of unit energy and right-angle scattering. The matrix element, as it
happens, is equal to e^2. (Note that are no conversion factors applied, so this
result is exact.)

For GNU make, `makeflags` is set to `-j1`. This eliminates a potential clash
with a `-j<n>` flag if this test is called from a parallel make.

⟨*Template matrix elements: execute tests*⟩≡
```
    call test (prc_template_me_1, "prc_template_me_1", &
        "build and load simple template process", &
        u, results)
```

⟨*Template matrix elements: tests*⟩+≡
```
    subroutine prc_template_me_1 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
```

```
class(prc_core_def_t), allocatable :: def
type(process_def_entry_t), pointer :: entry
type(os_data_t) :: os_data
type(model_list_t) :: model_list
type(model_t), pointer :: model
type(string_t) :: model_name
type(string_t), dimension(:), allocatable :: prt_in, prt_out
type(process_constants_t) :: data
class(prc_core_driver_t), allocatable :: driver
integer, parameter :: cdf = c_default_float
integer, parameter :: ci = c_int
real(cdf), dimension(4) :: par
real(cdf), dimension(0:3,4) :: p
logical(c_bool) :: flag
complex(c_default_complex) :: amp
integer :: i

write (u, "(A)")  "* Test output: prc_template_me_1"
write (u, "(A)")  "*   Purpose: create a template matrix element,"
write (u, "(A)")  "*             normalized to give unit integral,"
write (u, "(A)")  "*             build a library, link, load, and &
    &access the matrix element"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library with one entry"
write (u, "(A)")
call lib%init (var_str ("template_me1"))
call os_data_init (os_data)
call syntax_model_file_init ()

model => null ()
call model_list%read_model (var_str ("QED"), var_str ("QED.mdl"), &
    os_data, model)
model_name = model%get_name ()
allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e+"), var_str ("e-")]
prt_out = [var_str ("m+"), var_str ("m-")]

allocate (template_me_def_t :: def)
select type (def)
type is (template_me_def_t)
   call def%init (model, prt_in, prt_out, unity = .false.)
end select
allocate (entry)
call entry%init (var_str ("template_me1_a"), model_name = model_name, &
    n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
    prt_in  = new_prt_spec (prt_in), &
    prt_out = new_prt_spec (prt_out), &
    method  = var_str ("template"), &
    variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
```

2475

```
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of template_me1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("template_me1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,I0)")  "n_in  = ", data%n_in
write (u, "(1x,A,I0)")  "n_out = ", data%n_out
write (u, "(1x,A,I0)")  "n_flv = ", data%n_flv
write (u, "(1x,A,I0)")  "n_hel = ", data%n_hel
write (u, "(1x,A,I0)")  "n_col = ", data%n_col
write (u, "(1x,A,I0)")  "n_cin = ", data%n_cin
write (u, "(1x,A,I0)")  "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))")  "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))")  "hel state =", data%hel_state(:,1)
do i = 2, 16
   write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
end do
write (u, "(1x,A,10(1x,I0))")  "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))")  "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))")  "color factors =", data%color_factors
```

```
      write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

      write (u, "(A)")
      write (u, "(A)")   "* Set parameters for template_me1_a and initialize:"
      write (u, "(A)")

      par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
      write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
      write (u, "(2x,A,F6.4)")  "me   = ", par(2)
      write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
      write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

      write (u, "(A)")
      write (u, "(A)")   "* Set kinematics:"
      write (u, "(A)")

      p = reshape ([ &
           1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
           1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
           1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
           1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
           ], [4,4])
      do i = 1, 4
         write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
      end do

      select type (driver)
      type is (template_me_driver_t)
         call driver%init (par)

         call driver%new_event (p)

         write (u, "(A)")
         write (u, "(A)")   "* Compute matrix element:"
         write (u, "(A)")

         call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
         write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

         call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
         write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
      end select

      call lib%final ()
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: prc_template_me_1"

    end subroutine prc_template_me_1
```

⟨*Template matrix elements: execute tests*⟩+≡

```
      call test (prc_template_me_2, "prc_template_me_2", &
```

```
              "build and load simple template_unity process", &
              u, results)
```
⟨*Template matrix elements: tests*⟩+≡
```
        subroutine prc_template_me_2 (u)
          integer, intent(in) :: u
          type(process_library_t) :: lib
          class(prc_core_def_t), allocatable :: def
          type(process_def_entry_t), pointer :: entry
          type(os_data_t) :: os_data
          type(model_list_t) :: model_list
          type(model_t), pointer :: model
          type(string_t) :: model_name
          type(string_t), dimension(:), allocatable :: prt_in, prt_out
          type(process_constants_t) :: data
          class(prc_core_driver_t), allocatable :: driver
          integer, parameter :: cdf = c_default_float
          integer, parameter :: ci = c_int
          real(cdf), dimension(4) :: par
          real(cdf), dimension(0:3,4) :: p
          logical(c_bool) :: flag
          complex(c_default_complex) :: amp
          integer :: i

          write (u, "(A)")  "* Test output: prc_template_me_1"
          write (u, "(A)")  "*   Purpose: create a template matrix element,"
          write (u, "(A)")  "*            being identical to unity,"
          write (u, "(A)")  "*            build a library, link, load, and &
              &access the matrix element"
          write (u, "(A)")

          write (u, "(A)")  "* Initialize a process library with one entry"
          write (u, "(A)")
          call lib%init (var_str ("template_me2"))
          call os_data_init (os_data)
          call syntax_model_file_init ()

          model => null ()
          call model_list%read_model (var_str ("QED"), var_str ("QED.mdl"), &
              os_data, model)
          model_name = model%get_name ()
          allocate (prt_in (2), prt_out (2))
          prt_in = [var_str ("e+"), var_str ("e-")]
          prt_out = [var_str ("m+"), var_str ("m-")]

          allocate (template_me_def_t :: def)
          select type (def)
          type is (template_me_def_t)
             call def%init (model, prt_in, prt_out, unity = .true.)
          end select
          allocate (entry)
          call entry%init (var_str ("template_me2_a"), model_name = model_name, &
              n_in = 2, n_components = 1)
          call entry%import_component (1, n_out = size (prt_out), &
              prt_in  = new_prt_spec (prt_in), &
```

2478

```
      prt_out = new_prt_spec (prt_out), &
      method = var_str ("template_unity"), &
      variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of template_me2_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("template_me2_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
```

```
do i = 2, 16
   write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
end do
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

write (u, "(A)")
write (u, "(A)")  "* Set parameters for template_me2_a and initialize:"
write (u, "(A)")

par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

select type (driver)
type is (template_me_driver_t)
   call driver%init (par)

   call driver%new_event (p)

   write (u, "(A)")
   write (u, "(A)")  "* Compute matrix element:"
   write (u, "(A)")

   call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
   write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
   write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
end select

call lib%final ()
call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
```

2480

```
      write (u, "(A)")  "* Test output end: prc_template_me_2"

    end subroutine prc_template_me_2
```

18.2   `O'MEGA` Interface

The standard method for process computation with `WHIZARD` is the `O'MEGA`
matrix element generator.

   This section implements the interface to the code generator (via the makefile)
and the driver for the features provided by the `O'MEGA` matrix element.

⟨`prc_omega.f90`⟩≡
  ⟨*File header*⟩

```
  module prc_omega

    use iso_c_binding !NODEP!
    use kinds !NODEP!
```
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
```
    use limits, only: TAB !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use os_interface
    use lorentz !NODEP!
    use sm_qcd
    use interactions
    use variables
    use models

    use process_constants
    use prclib_interfaces
    use prc_core_def
    use particle_specifiers
    use process_libraries
    use prc_core
```

  ⟨*Standard module head*⟩

  ⟨*Omega interface: public*⟩

  ⟨*Omega interface: types*⟩

  ⟨*Omega interface: interfaces*⟩

```
  contains
```

  ⟨*Omega interface: procedures*⟩

  ⟨*Omega interface: tests*⟩

```
  end module prc_omega
```

18.2.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

⟨*Omega interface: public*⟩≡
```
public :: omega_def_t
```

⟨*Omega interface: types*⟩≡
```
type, extends (prc_core_def_t) :: omega_def_t
 contains
 ⟨Omega interface: omega def: TBP⟩
end type omega_def_t
```

⟨*Omega interface: omega def: TBP*⟩≡
```
procedure, nopass :: type_string => omega_def_type_string
```

⟨*Omega interface: procedures*⟩≡
```
function omega_def_type_string () result (string)
  type(string_t) :: string
  string = "omega"
end function omega_def_type_string
```

Initialization: allocate the writer for the O'MEGA matrix element. Also set any data for this process that the writer needs.

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure :: init => omega_def_init
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_init (object, model_name, prt_in, prt_out, &
    restrictions, openmp_support, report_progress, extra_options, &
    diags, diags_color)
  class(omega_def_t), intent(out) :: object
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  type(string_t), intent(in), optional :: restrictions
  logical, intent(in), optional :: openmp_support
  logical, intent(in), optional :: report_progress
  logical, intent(in), optional :: diags, diags_color
  type(string_t), intent(in), optional :: extra_options
  allocate (omega_writer_t :: object%writer)
  select type (writer => object%writer)
  type is (omega_writer_t)
     call writer%init (model_name, prt_in, prt_out, &
          restrictions, openmp_support, report_progress, &
          extra_options, diags, diags_color)
  end select
end subroutine omega_def_init
```

Write/read process- and method-specific data.

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure :: write => omega_def_write
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_write (object, unit)
  class(omega_def_t), intent(in) :: object
  integer, intent(in) :: unit
  select type (writer => object%writer)
  type is (omega_writer_t)
     call writer%write (unit)
  end select
end subroutine omega_def_write
```

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure :: read => omega_def_read
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_read (object, unit)
  class(omega_def_t), intent(out) :: object
  integer, intent(in) :: unit
  call msg_bug ("O'Mega process definition: input not supported yet")
end subroutine omega_def_read
```

Allocate the driver for O'MEGAmatrix elements.
⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure :: allocate_driver => omega_def_allocate_driver
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_allocate_driver (object, driver, basename)
  class(omega_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (omega_driver_t :: driver)
end subroutine omega_def_allocate_driver
```

We need code:
⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure, nopass :: needs_code => omega_def_needs_code
```

⟨*Omega interface: procedures*⟩+≡
```
function omega_def_needs_code () result (flag)
  logical :: flag
  flag = .true.
end function omega_def_needs_code
```

These are the features that an O'MEGA matrix element provides.
⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure, nopass :: get_features => omega_def_get_features
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (6))
  features = [ &
       var_str ("init"), &
       var_str ("update_alpha_s"), &
```

```
              var_str ("reset_helicity_selection"), &
              var_str ("is_allowed"), &
              var_str ("new_event"), &
              var_str ("get_amplitude")]
     end subroutine omega_def_get_features
```

The interface of the specific features.

⟨*Omega interface: interfaces*⟩≡

```
     abstract interface
        subroutine init_t (par) bind(C)
          import
          real(c_default_float), dimension(*), intent(in) :: par
        end subroutine init_t
     end interface

     abstract interface
        subroutine update_alpha_s_t (alpha_s) bind(C)
          import
          real(c_default_float), intent(in) :: alpha_s
        end subroutine update_alpha_s_t
     end interface

     abstract interface
        subroutine reset_helicity_selection_t (threshold, cutoff) bind(C)
          import
          real(c_default_float), intent(in) :: threshold
          integer(c_int), intent(in) :: cutoff
        end subroutine reset_helicity_selection_t
     end interface

     abstract interface
        subroutine is_allowed_t (flv, hel, col, flag) bind(C)
          import
          integer(c_int), intent(in) :: flv, hel, col
          logical(c_bool), intent(out) :: flag
        end subroutine is_allowed_t
     end interface

     abstract interface
        subroutine new_event_t (p) bind(C)
          import
          real(c_default_float), dimension(0:3,*), intent(in) :: p
        end subroutine new_event_t
     end interface

     abstract interface
        subroutine get_amplitude_t (flv, hel, col, amp) bind(C)
          import
          integer(c_int), intent(in) :: flv, hel, col
          complex(c_default_complex), intent(out):: amp
        end subroutine get_amplitude_t
     end interface
```

Connect the `O'MEGA` features with the process driver.

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure :: connect => omega_def_connect
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_connect (def, lib_driver, i, proc_driver)
  class(omega_def_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  integer(c_int) :: pid, fid
  type(c_funptr) :: fptr
  select type (proc_driver)
  type is  (omega_driver_t)
     pid = i
     fid = 1
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%init)
     fid = 2
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%update_alpha_s)
     fid = 3
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%reset_helicity_selection)
     fid = 4
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%is_allowed)
     fid = 5
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%new_event)
     fid = 6
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%get_amplitude)
  end select
end subroutine omega_def_connect
```

18.2.2   The `O'MEGA` writer

The `O'MEGA` writer is responsible for inserting the appropriate lines in the makefile that call `O'MEGA`, and for writing interfaces and wrappers.

⟨*Omega interface: types*⟩+≡
```
type, extends (prc_writer_f_module_t) :: omega_writer_t
  type(string_t) :: model_name
  type(string_t) :: process_mode
  type(string_t) :: process_string
  type(string_t) :: restrictions
  logical :: openmp_support = .false.
  logical :: report_progress = .false.
  logical :: diags = .false.
  logical :: diags_color = .false.
  type(string_t) :: extra_options
contains
```

⟨*Omega interface: omega writer: TBP*⟩
```
    end type omega_writer_t
```

The reported type is the same as for the `omega_def_t` type.

⟨*Omega interface: omega writer: TBP*⟩≡
```
    procedure, nopass :: type_name => omega_writer_type_name
```

⟨*Omega interface: procedures*⟩+≡
```
    function omega_writer_type_name () result (string)
      type(string_t) :: string
      string = "omega"
    end function omega_writer_type_name
```

Taking into account the prefix for `O'MEGA` module names.

⟨*Omega interface: omega writer: TBP*⟩+≡
```
    procedure, nopass :: get_module_name => omega_writer_get_module_name
```

⟨*Omega interface: procedures*⟩+≡
```
    function omega_writer_get_module_name (id) result (name)
      type(string_t) :: name
      type(string_t), intent(in) :: id
      name = "opr_" // id
    end function omega_writer_get_module_name
```

Output. This is called by `omega_def_write`.

⟨*Omega interface: omega writer: TBP*⟩+≡
```
    procedure :: write => omega_writer_write
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine omega_writer_write (object, unit)
      class(omega_writer_t), intent(in) :: object
      integer, intent(in) :: unit
      write (unit, "(5x,A,A)")  "Model name       = ", &
           '"' // char (object%model_name) // '"'
      write (unit, "(5x,A,A)")  "Mode string      = ", &
           '"' // char (object%process_mode) // '"'
      write (unit, "(5x,A,A)")  "Process string   = ", &
           '"' // char (object%process_string) // '"'
      write (unit, "(5x,A,A)")  "Restrictions     = ", &
           '"' // char (object%restrictions) // '"'
      write (unit, "(5x,A,L1)")  "OpenMP support    = ", object%openmp_support
      write (unit, "(5x,A,L1)")  "Report progress   = ", object%report_progress
      write (unit, "(5x,A,A)")  "Extra options    = ", &
           '"' // char (object%extra_options) // '"'
      write (unit, "(5x,A,L1)")  "Write diagrams    = ", object%diags
      write (unit, "(5x,A,L1)")  "Write color diag. = ", object%diags_color
    end subroutine omega_writer_write
```

Initialize with process data.

⟨*Omega interface: omega writer: TBP*⟩+≡
```
    procedure :: init => omega_writer_init
```

⟨Omega interface: procedures⟩+≡

```
    subroutine omega_writer_init (writer, model_name, prt_in, prt_out, &
        restrictions, openmp_support, report_progress, extra_options, &
        diags, diags_color)
  class(omega_writer_t), intent(out) :: writer
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  type(string_t), intent(in), optional :: restrictions
  logical, intent(in), optional :: openmp_support
  logical, intent(in), optional :: report_progress
  logical, intent(in), optional :: diags, diags_color
  type(string_t), intent(in), optional :: extra_options
  integer :: i
  writer%model_name = model_name
  if (present (restrictions)) then
     writer%restrictions = restrictions
  else
     writer%restrictions = ""
  end if
  if (present (openmp_support))  writer%openmp_support = openmp_support
  if (present (report_progress))  writer%report_progress = report_progress
  if (present (diags))  writer%diags = diags
  if (present (diags_color))  writer%diags_color = diags_color
  if (present (extra_options)) then
     writer%extra_options = " " // extra_options
  else
     writer%extra_options = ""
  end if
  select case (size (prt_in))
  case (1);  writer%process_mode = " -decay"
  case (2);  writer%process_mode = " -scatter"
  end select
  associate (s => writer%process_string)
    s = " '"
    do i = 1, size (prt_in)
       if (i > 1)  s = s // " "
       s = s // prt_in(i)
    end do
    s = s // " ->"
    do i = 1, size (prt_out)
       s = s // " " // prt_out(i)
    end do
    s = s // "'"
  end associate
end subroutine omega_writer_init
```

The makefile implements the actual O'MEGA call. For old LaTeX distributions,
we filter out the hyperref options for O'MEGA diagrams, at least in the testsuite.

⟨Omega interface: omega writer: TBP⟩+≡

```
    procedure :: write_makefile_code => omega_write_makefile_code
```

⟨Omega interface: procedures⟩+≡

```
    subroutine omega_write_makefile_code (writer, unit, id, os_data, testflag)
```

```
class(omega_writer_t), intent(in) :: writer
integer, intent(in) :: unit
type(string_t), intent(in) :: id
type(os_data_t), intent(in) :: os_data
logical, intent(in), optional :: testflag
type(string_t) :: omega_binary, omega_path
type(string_t) :: restrictions_string
type(string_t) :: openmp_string
type(string_t) :: kmatrix_string
type(string_t) :: progress_string
type(string_t) :: diagrams_string
logical :: escape_hyperref
escape_hyperref = .false.
if (present (testflag))  escape_hyperref = testflag
omega_binary = "omega_" // writer%model_name // ".opt"
omega_path = os_data%whizard_omega_binpath // "/" // omega_binary
if (writer%restrictions /= "") then
   restrictions_string = " -cascade '" // writer%restrictions // "'"
else
   restrictions_string = ""
end if
if (writer%openmp_support) then
   openmp_string = " -target:openmp"
else
   openmp_string = ""
end if
if (writer%report_progress) then
   progress_string = " -fusion:progress"
else
   progress_string = ""
end if
if (writer%diags) then
   if (writer%diags_color) then
      diagrams_string = " -diagrams:C " // char(id) // &
           "_diags -diagrams_LaTeX"
   else
      diagrams_string = " -diagrams " // char(id) // &
           "_diags -diagrams_LaTeX"
   end if
else
   if (writer%diags_color) then
      diagrams_string = " -diagrams:c " // char(id) // &
           "_diags -diagrams_LaTeX"
   else
      diagrams_string = ""
   end if
end if
select case (char (writer%model_name))
case ("SM_rx", "SSC", "NoH_rx", "AltH")
   kmatrix_string = " -target:kmatrix_write"
case default
   kmatrix_string = ""
end select
write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
```

```
if (writer%diags .or. writer%diags_color) then
   write (unit, "(5A)")  "TEX_SOURCES += ", char (id), "_diags.tex"
   if (os_data%event_analysis_pdf) then
      write (unit, "(5A)")  "TEX_OBJECTS += ", char (id), "_diags.pdf"
   else
      write (unit, "(5A)")  "TEX_OBJECTS += ", char (id), "_diags.ps"
   end if
end if
write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
write (unit, "(5A)")  char (id), ".f90:"
write (unit, "(99A)")  TAB, char (omega_path), &
     " -o ", char (id), ".f90", &
     " -target:whizard", &
     " -target:parameter_module parameters_", char (writer%model_name), &
     " -target:module opr_", char (id), &
     " -target:md5sum '", writer%md5sum, "'", &
     char (openmp_string), &
     char (progress_string), &
     char (kmatrix_string), &
     char (writer%process_mode), char (writer%process_string), &
     char (restrictions_string), char (diagrams_string), &
     char (writer%extra_options)
if (writer%diags .or. writer%diags_color) &
   write (unit, "(5A)")  char (id), "_diags.tex: ", char (id), ".f90"
write (unit, "(5A)")  "clean-", char (id), ":"
write (unit, "(5A)")  TAB, "rm -f ", char (id), ".f90"
write (unit, "(5A)")  TAB, "rm -f opr_", char (id), ".mod"
write (unit, "(5A)")  TAB, "rm -f ", char (id), ".lo"
write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
if (writer%diags .or. writer%diags_color) then
   write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), "_diags.tex"
end if
write (unit, "(5A)")  "CLEAN_OBJECTS += opr_", char (id), ".mod"
write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
if (writer%diags .or. writer%diags_color) then
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.aux"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.log"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.dvi"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.toc"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.out"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.[1-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.[1-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.[1-9][0-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.t[1-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.t[1-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.t[1-9][0-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.mp"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.log"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.dvi"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.ps"
   if (os_data%event_analysis_pdf) &
        write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.pdf"
end if
write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
```

2489

```
      write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
      if (writer%diags .or. writer%diags_color) then
         if (os_data%event_analysis_ps) then
            if (os_data%event_analysis_pdf) then
               write (unit, "(5A)")  char (id), "_diags.pdf: ", char (id), "_diags.tex"
            else
               write (unit, "(5A)")  char (id), "_diags.ps: ", char (id), "_diags.tex"
            end if
            if (escape_hyperref) then
               write (unit, "(5A)")  TAB, "-cat ", char (id), "_diags.tex | \"
               write (unit, "(5A)")  TAB, "   sed -e" // &
                  "'s/\\usepackage\[colorlinks\]{hyperref}.*/%\\usepackage" // &
                  "\[colorlinks\]{hyperref}/' > \"
               write (unit, "(5A)")  TAB, "   ", char (id), "_diags.tex.tmp"
               write (unit, "(5A)")  TAB, "mv -f ", char (id), "_diags.tex.tmp \"
               write (unit, "(5A)")  TAB, "   ", char (id), "_diags.tex"
            end if
            write (unit, "(5A)")  TAB, "-TEXINPUTS=$(TEX_FLAGS) $(LATEX) " // &
               char (id) // "_diags.tex"
            write (unit, "(5A)")  TAB, "MPINPUTS=$(MP_FLAGS) $(MPOST) " // &
               char (id) // "_diags-fmf.mp"
            write (unit, "(5A)")  TAB, "TEXINPUTS=$(TEX_FLAGS) $(LATEX) " // &
               char (id) // "_diags.tex"
            write (unit, "(5A)")  TAB, "$(DVIPS) -o " // char (id) // "_diags.ps " // &
               char (id) // "_diags.dvi"
            if (os_data%event_analysis_pdf) then
               write (unit, "(5A)")  TAB, "$(PS2PDF) " // char (id) // "_diags.ps"
            end if
         end if
      end if
    end subroutine omega_write_makefile_code
```

The source is written by the makefile, so nothing to do here.

⟨Omega interface: omega writer: TBP⟩+≡

```
    procedure :: write_source_code => omega_write_source_code
```

⟨Omega interface: procedures⟩+≡

```
    subroutine omega_write_source_code (writer, id)
      class(omega_writer_t), intent(in) :: writer
      type(string_t), intent(in) :: id
    end subroutine omega_write_source_code
```

Return the name of a procedure that implements a given feature, as it is provided
by the external matrix-element code. O'MEGA names some procedures differently,
therefore we translate here and override the binding of the base type.

⟨Omega interface: omega writer: TBP⟩+≡

```
    procedure, nopass :: get_procname => omega_writer_get_procname
```

⟨Omega interface: procedures⟩+≡

```
    function omega_writer_get_procname (feature) result (name)
      type(string_t) :: name
      type(string_t), intent(in) :: feature
      select case (char (feature))
      case ("n_in");   name = "number_particles_in"
```

```
      case ("n_out");  name = "number_particles_out"
      case ("n_flv");  name = "number_flavor_states"
      case ("n_hel");  name = "number_spin_states"
      case ("n_col");  name = "number_color_flows"
      case ("n_cin");  name = "number_color_indices"
      case ("n_cf");   name = "number_color_factors"
      case ("flv_state"); name = "flavor_states"
      case ("hel_state"); name = "spin_states"
      case ("col_state"); name = "color_flows"
      case default
         name = feature
      end select
   end function omega_writer_get_procname
```

The interfaces for the O'MEGA-specific features.

⟨*Omega interface: omega writer: TBP*⟩+≡
```
    procedure :: write_interface => omega_write_interface
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine omega_write_interface (writer, unit, id, feature)
      class(omega_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id
      type(string_t), intent(in) :: feature
      type(string_t) :: name
      name = writer%get_c_procname (id, feature)
      write (unit, "(2x,9A)")  "interface"
      select case (char (feature))
      case ("init")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " (par) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), dimension(*), &
               &intent(in) :: par"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("update_alpha_s")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: alpha_s"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("reset_helicity_selection")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " &
               &(threshold, cutoff) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: threshold"
         write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: cutoff"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("is_allowed")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " &
               &(flv, hel, col, flag) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
         write (unit, "(7x,9A)")  "logical(c_bool), intent(out) :: flag"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("new_event")
```

```
         write (unit, "(5x,9A)")  "subroutine ", char (name), " (p) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "real(c_default_float), dimension(0:3,*), &
              &intent(in) :: p"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      case ("get_amplitude")
         write (unit, "(5x,9A)")  "subroutine ", char (name), " &
              &(flv, hel, col, amp) bind(C)"
         write (unit, "(7x,9A)")  "import"
         write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
         write (unit, "(7x,9A)")  "complex(c_default_complex), intent(out) &
              &:: amp"
         write (unit, "(5x,9A)")  "end subroutine ", char (name)
      end select
      write (unit, "(2x,9A)")  "end interface"
   end subroutine omega_write_interface
```

The wrappers have to take into account conversion between C and Fortran data
types.

NOTE: The case $c\_default\_float \neq default$ is not yet covered.

⟨*Omega interface: omega writer: TBP*⟩+≡
```
    procedure :: write_wrapper => omega_write_wrapper
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine omega_write_wrapper (writer, unit, id, feature)
      class(omega_writer_t), intent(in) :: writer
      integer, intent(in) :: unit
      type(string_t), intent(in) :: id, feature
      type(string_t) :: name
      name = writer%get_c_procname (id, feature)
      write (unit, *)
      select case (char (feature))
      case ("init")
         write (unit, "(9A)")  "subroutine ", char (name), " (par) bind(C)"
         write (unit, "(2x,9A)")  "use iso_c_binding"
         write (unit, "(2x,9A)")  "use kinds"
         write (unit, "(2x,9A)")  "use opr_", char (id)
         write (unit, "(2x,9A)")  "real(c_default_float), dimension(*), &
              &intent(in) :: par"
         if (c_default_float == default) then
            write (unit, "(2x,9A)")  "call ", char (feature), " (par)"
         end if
         write (unit, "(9A)")  "end subroutine ", char (name)
      case ("update_alpha_s")
         write (unit, "(9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
         write (unit, "(2x,9A)")  "use iso_c_binding"
         write (unit, "(2x,9A)")  "use kinds"
         write (unit, "(2x,9A)")  "use opr_", char (id)
         if (c_default_float == default) then
            write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
                 &:: alpha_s"
            write (unit, "(2x,9A)")  "call ", char (feature), " (alpha_s)"
         end if
         write (unit, "(9A)")  "end subroutine ", char (name)
```

```
     case ("reset_helicity_selection")
        write (unit, "(9A)")  "subroutine ", char (name), &
             " (threshold, cutoff) bind(C)"
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use opr_", char (id)
        if (c_default_float == default) then
           write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
                &:: threshold"
           write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: cutoff"
           write (unit, "(2x,9A)")  "call ", char (feature), &
                " (threshold, int (cutoff))"
        end if
        write (unit, "(9A)")  "end subroutine ", char (name)
  case ("is_allowed")
        write (unit, "(9A)")  "subroutine ", char (name), &
             " (flv, hel, col, flag) bind(C)"
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use opr_", char (id)
        write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
        write (unit, "(2x,9A)")  "logical(c_bool), intent(out) :: flag"
        write (unit, "(2x,9A)")  "flag = ", char (feature), &
             " (int (flv), int (hel), int (col))"
        write (unit, "(9A)")  "end subroutine ", char (name)
  case ("new_event")
        write (unit, "(9A)")  "subroutine ", char (name), " (p) bind(C)"
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use opr_", char (id)
        if (c_default_float == default) then
           write (unit, "(2x,9A)")  "real(c_default_float), dimension(0:3,*), &
                &intent(in) :: p"
           write (unit, "(2x,9A)")  "call ", char (feature), " (p)"
        end if
        write (unit, "(9A)")  "end subroutine ", char (name)
  case ("get_amplitude")
        write (unit, "(9A)")  "subroutine ", char (name), &
             " (flv, hel, col, amp) bind(C)"
        write (unit, "(2x,9A)")  "use iso_c_binding"
        write (unit, "(2x,9A)")  "use kinds"
        write (unit, "(2x,9A)")  "use opr_", char (id)
        write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
        write (unit, "(2x,9A)")  "complex(c_default_complex), intent(out) &
             &:: amp"
        write (unit, "(2x,9A)")  "amp = ", char (feature), &
             " (int (flv), int (hel), int (col))"
        write (unit, "(9A)")  "end subroutine ", char (name)
  end select
end subroutine omega_write_wrapper
```

18.2.3 Driver

⟨Omega interface: types⟩+≡
```
type, extends (prc_core_driver_t) :: omega_driver_t
  procedure(init_t), nopass, pointer :: &
       init => null ()
  procedure(update_alpha_s_t), nopass, pointer :: &
       update_alpha_s => null ()
  procedure(reset_helicity_selection_t), nopass, pointer :: &
       reset_helicity_selection => null ()
  procedure(is_allowed_t), nopass, pointer :: &
       is_allowed => null ()
  procedure(new_event_t), nopass, pointer :: &
       new_event => null ()
  procedure(get_amplitude_t), nopass, pointer :: &
       get_amplitude => null ()
 contains
 ⟨Omega interface: omega driver: TBP⟩
end type omega_driver_t
```

The reported type is the same as for the `omega_def_t` type.

⟨Omega interface: omega driver: TBP⟩≡
```
procedure, nopass :: type_name => omega_driver_type_name
```

⟨Omega interface: procedures⟩+≡
```
function omega_driver_type_name () result (string)
  type(string_t) :: string
  string = "omega"
end function omega_driver_type_name
```

18.2.4 High-level process definition

This procedure wraps the details filling a process-component definition entry as appropriate for an `O'MEGA` matrix element.

NOTE: For calling the `import_component` method, we must explicitly address the `process_def_t` parent object. The natural way to call the method of the extended type triggers a bug in gfortran 4.6. The string array arguments `prt_in` and `prt_out` become corrupted and cause a segfault.

⟨Omega interface: public⟩+≡
```
public :: omega_make_process_component
```

⟨Omega interface: procedures⟩+≡
```
subroutine omega_make_process_component (entry, component_index, &
     model_name, prt_in, prt_out, restrictions, openmp_support, &
     report_progress, extra_options, diags, diags_color)
  class(process_def_entry_t), intent(inout) :: entry
  integer, intent(in) :: component_index
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  type(string_t), intent(in), optional :: restrictions
  logical, intent(in), optional :: openmp_support
  logical, intent(in), optional :: report_progress
```

```
      logical, intent(in), optional :: diags, diags_color
      type(string_t), intent(in), optional :: extra_options
      class(prc_core_def_t), allocatable :: def
      allocate (omega_def_t :: def)
      select type (def)
      type is (omega_def_t)
         call def%init (model_name, prt_in, prt_out, &
              restrictions, openmp_support, report_progress, &
              extra_options, diags, diags_color)
      end select
      call entry%process_def_t%import_component (component_index, &
           n_out = size (prt_out), &
           prt_in  = new_prt_spec (prt_in), &
           prt_out = new_prt_spec (prt_out), &
           method = var_str ("omega"), &
           variant = def)
    end subroutine omega_make_process_component
```

18.2.5 The `prc_omega_t` wrapper

This is an instance of the generic `prc_core_t` object. It contains a pointer to the process definition (`omega_def_t`), a data component (`process_constants_t`), and the matrix-element driver (`omega_driver_t`).

⟨*Omega interface: public*⟩+≡
```
    public :: prc_omega_t
```

⟨*Omega interface: types*⟩+≡
```
    type, extends (prc_core_t) :: prc_omega_t
      real(default), dimension(:), allocatable :: par
      type(helicity_selection_t) :: helicity_selection
      type(qcd_t) :: qcd
     contains
     ⟨Omega interface: prc omega: TBP⟩
    end type prc_omega_t
```

The workspace associated to a `prc_omega_t` object contains a single flag. The flag is used to suppress re-evaluating the matrix element for each quantum-number combination, after the first amplitude belonging to a given kinematics has been computed.

We can also store the value of a running coupling once it has been calculated for an event. The default value is negative, which indicates an undefined value in this context.

⟨*Omega interface: types*⟩+≡
```
    type, extends (workspace_t) :: omega_state_t
      logical :: new_kinematics = .true.
      real(default) :: alpha_qcd = -1
     contains
       procedure :: write => omega_state_write
    end type omega_state_t
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine omega_state_write (object, unit)
      class(omega_state_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(3x,A,L1)")  "O'Mega state: new kinematics = ", &
           object%new_kinematics
    end subroutine omega_state_write
```

Allocate the workspace with the above specific type.

⟨*Omega interface: prc omega: TBP*⟩≡
```
    procedure :: allocate_workspace => prc_omega_allocate_workspace
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine prc_omega_allocate_workspace (object, tmp)
      class(prc_omega_t), intent(in) :: object
      class(workspace_t), intent(inout), allocatable :: tmp
      allocate (omega_state_t :: tmp)
    end subroutine prc_omega_allocate_workspace
```

The following procedures are inherited from the base type as deferred, thus must be implemented. The corresponding unit tests are skipped here; the procedures are tested when called from the `processes` module.

Output: print just the ID of the associated matrix element. Then display any stored parameters and the helicity selection data. (The latter are printed only if active.)

⟨*Omega interface: prc omega: TBP*⟩+≡
```
    procedure :: write => prc_omega_write
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine prc_omega_write (object, unit)
      class(prc_omega_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(3x,A)", advance="no")  "O'Mega process core:"
      if (object%data_known) then
         write (u, "(1x,A)")  char (object%data%id)
      else
         write (u, "(1x,A)")  "[undefined]"
      end if
      if (allocated (object%par)) then
         write (u, "(3x,A)")  "Parameter array:"
         do i = 1, size (object%par)
            write (u, "(5x,I0,1x,ES17.10)")  i, object%par(i)
         end do
      end if
      call object%helicity_selection%write (u)
      call object%qcd%write (u)
    end subroutine prc_omega_write
```

Temporarily store the parameter array inside the `prc_omega` object, so we can use it later during the actual initialization. Also store threshold and cutoff for helicity selection.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
    procedure :: set_parameters => prc_omega_set_parameters
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine prc_omega_set_parameters (prc_omega, model, &
        helicity_selection, qcd, use_color_factors)
      class(prc_omega_t), intent(inout) :: prc_omega
      type(model_t), intent(in), target, optional :: model
      type(helicity_selection_t), intent(in), optional :: helicity_selection
      type(qcd_t), intent(in), optional :: qcd
      logical, intent(in), optional :: use_color_factors
      if (present (model)) then
         if (allocated (prc_omega%par))  deallocate (prc_omega%par)
         call model_parameters_to_c_array (model, prc_omega%par)
      end if
      if (present (helicity_selection)) then
         prc_omega%helicity_selection = helicity_selection
      end if
      if (present (qcd)) then
         prc_omega%qcd = qcd
      end if
      if (present (use_color_factors)) then
         prc_omega%use_color_factors = use_color_factors
      end if
    end subroutine prc_omega_set_parameters
```

To fully initialize the process core, we perform base initialization, then initialize the external matrix element code.

This procedure overrides the `init` method of the base type, which we nevertheless can access via its binding `base_init`. When done, we have an allocated driver. The driver will call the `init` procedure for the external matrix element, and thus transfer the parameter set to where it finally belongs.

If requested, we initialize the helicity selction counter.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
    procedure :: init => prc_omega_init
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine prc_omega_init (object, def, lib, id, i_component)
      class(prc_omega_t), intent(inout) :: object
      class(prc_core_def_t), intent(in), target :: def
      type(process_library_t), intent(in), target :: lib
      type(string_t), intent(in) :: id
      integer, intent(in) :: i_component
      call object%base_init (def, lib, id, i_component)
      call object%activate_parameters ()
    end subroutine prc_omega_init
```

Activate the stored parameters by transferring them to the external matrix element. Also reset the helicity selection, if requested.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
      procedure :: activate_parameters => prc_omega_activate_parameters
```

⟨*Omega interface: procedures*⟩+≡
```
    subroutine prc_omega_activate_parameters (object)
      class (prc_omega_t), intent(inout) :: object
      if (allocated (object%driver)) then
         if (allocated (object%par)) then
            select type (driver => object%driver)
            type is (omega_driver_t)
               if (associated (driver%init))  call driver%init (object%par)
            end select
         else
            call msg_bug ("prc_omega_activate: parameter set is not allocated")
         end if
         call object%reset_helicity_selection ()
      else
         call msg_bug ("prc_omega_activate: driver is not allocated")
      end if
    end subroutine prc_omega_activate_parameters
```

The `O'MEGA` process is an independent process which needs its own Monte Carlo parameter set for integration.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
    procedure :: needs_mcset => prc_omega_needs_mcset
```

⟨*Omega interface: procedures*⟩+≡
```
    function prc_omega_needs_mcset (object) result (flag)
      class(prc_omega_t), intent(in) :: object
      logical :: flag
      flag = .true.
    end function prc_omega_needs_mcset
```

There is only one term for this tree-level process.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
    procedure :: get_n_terms => prc_omega_get_n_terms
```

⟨*Omega interface: procedures*⟩+≡
```
    function prc_omega_get_n_terms (object) result (n)
      class(prc_omega_t), intent(in) :: object
      integer :: n
      n = 1
    end function prc_omega_get_n_terms
```

Tell whether a particular combination of flavor, helicity, color is allowed. Here we have to consult the matrix-element driver.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
    procedure :: is_allowed => prc_omega_is_allowed
```

⟨*Omega interface: procedures*⟩+≡
```
  function prc_omega_is_allowed (object, i_term, f, h, c) result (flag)
    class(prc_omega_t), intent(in) :: object
    integer, intent(in) :: i_term, f, h, c
    logical :: flag
    logical(c_bool) :: cflag
```

2498

```
        select type (driver => object%driver)
        type is (omega_driver_t)
           call driver%is_allowed (f, h, c, cflag)
           flag = cflag
        end select
     end function prc_omega_is_allowed
```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

We reset the `new_kinematics` flag, so that the next call to `compute_amplitude` will evaluate the matrix element.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
     procedure :: compute_hard_kinematics => prc_omega_compute_hard_kinematics
```

⟨*Omega interface: procedures*⟩+≡
```
     subroutine prc_omega_compute_hard_kinematics &
         (object, p_seed, i_term, int_hard, tmp)
       class(prc_omega_t), intent(in) :: object
       type(vector4_t), dimension(:), intent(in) :: p_seed
       integer, intent(in) :: i_term
       type(interaction_t), intent(inout) :: int_hard
       class(workspace_t), intent(inout), allocatable :: tmp
       call interaction_set_momenta (int_hard, p_seed)
       if (allocated (tmp)) then
          select type (tmp)
          type is (omega_state_t);  tmp%new_kinematics = .true.
          end select
       end if
     end subroutine prc_omega_compute_hard_kinematics
```

This procedure is not called for `prc_omega_t`, just a placeholder.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
     procedure :: compute_eff_kinematics => prc_omega_compute_eff_kinematics
```

⟨*Omega interface: procedures*⟩+≡
```
     subroutine prc_omega_compute_eff_kinematics &
         (object, i_term, int_hard, int_eff, tmp)
       class(prc_omega_t), intent(in) :: object
       integer, intent(in) :: i_term
       type(interaction_t), intent(in) :: int_hard
       type(interaction_t), intent(inout) :: int_eff
       class(workspace_t), intent(inout), allocatable :: tmp
     end subroutine prc_omega_compute_eff_kinematics
```

Recover the momenta within the hard interaction. For tree-level processes, this is a trivial copy of the incoming seed and outgoing effective momenta. The effective interaction is a pointer to the hard interaction, so the latter is ignored.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
     procedure :: recover_kinematics => prc_omega_recover_kinematics
```

⟨*Omega interface: tests*⟩≡
```
     subroutine prc_omega_recover_kinematics &
         (object, p_seed, int_hard, int_eff, tmp)
```

```
        class(prc_omega_t), intent(in) :: object
        type(vector4_t), dimension(:), intent(inout) :: p_seed
        type(interaction_t), intent(inout) :: int_hard
        type(interaction_t), intent(inout) :: int_eff
        class(workspace_t), intent(inout), allocatable :: tmp
        integer :: n_in
        n_in = interaction_get_n_in (int_eff)
        call interaction_set_momenta (int_eff, p_seed(1:n_in), outgoing = .false.)
        p_seed(n_in+1:) = interaction_get_momenta (int_eff, outgoing = .true.)
      end subroutine prc_omega_recover_kinematics
```

Reset the helicity selection counters and start counting zero helicities. We assume that the `helicity_selection` object is allocated. Otherwise, reset and switch off helicity counting.

In the test routine, the driver is allocated but the driver methods are not. Therefore, guard against a disassociated method.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
      procedure :: reset_helicity_selection => prc_omega_reset_helicity_selection
```

⟨*Omega interface: procedures*⟩+≡
```
      subroutine prc_omega_reset_helicity_selection (object)
        class(prc_omega_t), intent(inout) :: object
        select type (driver => object%driver)
        type is (omega_driver_t)
           if (associated (driver%reset_helicity_selection)) then
              if (object%helicity_selection%active) then
                 call driver%reset_helicity_selection &
                      (real (object%helicity_selection%threshold, &
                      c_default_float), &
                      int (object%helicity_selection%cutoff, c_int))
              else
                 call driver%reset_helicity_selection &
                      (0._c_default_float, 0_c_int)
              end if
           end if
        end select
      end subroutine prc_omega_reset_helicity_selection
```

Compute the amplitude. For the tree-level process, we can ignore the scale settings. The term index `j` is also irrelevant.

We first call `new_event` for the given momenta (which we must unpack), then retrieve the amplitude value for the given quantum numbers.

If the `tmp` status flag is present, we can make sure that we call `new_event` only once for a given kinematics. After the first call, we unset the `new_kinematics` flag.

The `is_allowed` query is not redundant, since the status may change during the run if helicities are switched off.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
      procedure :: compute_amplitude => prc_omega_compute_amplitude
```

⟨*Omega interface: procedures*⟩+≡
```
      function prc_omega_compute_amplitude &
           (object, j, p, f, h, c, fac_scale, ren_scale, tmp) result (amp)
```

```fortran
      class(prc_omega_t), intent(in) :: object
      integer, intent(in) :: j
      type(vector4_t), dimension(:), intent(in) :: p
      integer, intent(in) :: f, h, c
      real(default), intent(in) :: fac_scale, ren_scale
      class(workspace_t), intent(inout), allocatable, optional :: tmp
      real(default) :: alpha_qcd
      complex(default) :: amp
      integer :: n_tot, i
      real(c_default_float), dimension(:,:), allocatable :: parray
      complex(c_default_complex) :: camp
      logical :: new_event
      select type (driver => object%driver)
      type is (omega_driver_t)
         new_event = .true.
         if (present (tmp)) then
            if (allocated (tmp)) then
               select type (tmp)
               type is (omega_state_t)
                  new_event = tmp%new_kinematics
                  tmp%new_kinematics = .false.
               end select
            end if
         end if
         if (new_event) then
            if (allocated (object%qcd%alpha)) then
               alpha_qcd = object%qcd%alpha%get (fac_scale)
               call driver%update_alpha_s (alpha_qcd)
               if (present (tmp)) then
                  if (allocated (tmp)) then
                     select type (tmp)
                     type is (omega_state_t)
                        tmp%alpha_qcd = alpha_qcd
                     end select
                  end if
               end if
            end if
            n_tot = object%data%n_in + object%data%n_out
            allocate (parray (0:3, n_tot))
            do i = 1, n_tot
               parray(:,i) = vector4_get_components (p(i))
            end do
            call driver%new_event (parray)
         end if
         if (object%is_allowed (1, f, h, c)) then
            call driver%get_amplitude &
                 (int (f, c_int), int (h, c_int), int (c, c_int), camp)
            amp = camp
         else
            amp = 0
         end if
      end select
    end function prc_omega_compute_amplitude
```

After the amplitude has been computed, we may read off the current value of α_s. This works only if α_s varies, and if the workspace `tmp` is present which stores this value.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
procedure :: get_alpha_s => prc_omega_get_alpha_s
```

⟨*Omega interface: procedures*⟩+≡
```
function prc_omega_get_alpha_s (object, tmp) result (alpha)
  class(prc_omega_t), intent(in) :: object
  class(workspace_t), intent(in), allocatable :: tmp
  real(default) :: alpha
  alpha = -1
  if (allocated (object%qcd%alpha) .and. allocated (tmp)) then
    select type (tmp)
    type is (omega_state_t)
      alpha = tmp%alpha_qcd
    end select
  end if
end function prc_omega_get_alpha_s
```

18.2.6 Test

This is the master for calling self-test procedures. There is a separate test for testing O'MEGA diagram generation as this depends on a working analysis setup.

⟨*Omega interface: public*⟩+≡
```
public :: prc_omega_test
```

⟨*Omega interface: tests*⟩+≡
```
subroutine prc_omega_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
  ⟨Omega interface: execute tests⟩
end subroutine prc_omega_test
```

⟨*Omega interface: public*⟩+≡
```
public :: prc_omega_diags_test
```

⟨*Omega interface: tests*⟩+≡
```
subroutine prc_omega_diags_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
  ⟨Omega interface: execute diags tests⟩
end subroutine prc_omega_diags_test
```

Generate, compile and load a simple process matrix element

The process is $e^+e^- \rightarrow \mu^+\mu^-$ for vanishing masses and $e = 0.3$. We initialize the process, build the library, and compute a particular matrix element for momenta of unit energy and right-angle scattering. The matrix element, as it happens, is equal to e^2. (Note that are no conversion factors applied, so this result is exact.)

2502

For GNU make, makeflags is set to -j1. This eliminates a potential clash with a -j<n> flag if this test is called from a parallel make.

⟨*Omega interface: execute tests*⟩≡
```
call test (prc_omega_1, "prc_omega_1", &
     "build and load simple OMega process", &
     u, results)
```

⟨*Omega interface: tests*⟩+≡
```
subroutine prc_omega_1 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: driver
  integer, parameter :: cdf = c_default_float
  integer, parameter :: ci = c_int
  real(cdf), dimension(4) :: par
  real(cdf), dimension(0:3,4) :: p
  logical(c_bool) :: flag
  complex(c_default_complex) :: amp
  integer :: i

  write (u, "(A)")  "* Test output: prc_omega_1"
  write (u, "(A)")  "*   Purpose: create a simple process with OMega"
  write (u, "(A)")  "*            build a library, link, load, and &
       &access the matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("omega1"))
  call os_data_init (os_data)

  model_name = "QED"
  allocate (prt_in (2), prt_out (2))
  prt_in = [var_str ("e+"), var_str ("e-")]
  prt_out = [var_str ("m+"), var_str ("m-")]

  allocate (omega_def_t :: def)
  select type (def)
  type is (omega_def_t)
     call def%init (model_name, prt_in, prt_out)
  end select
  allocate (entry)
  call entry%init (var_str ("omega1_a"), model_name = model_name, &
       n_in = 2, n_components = 1)
  call entry%import_component (1, n_out = size (prt_out), &
       prt_in  = new_prt_spec (prt_in), &
       prt_out = new_prt_spec (prt_out), &
       method  = var_str ("omega"), &
```

```
      variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of omega1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("omega1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,L1)")  "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)")  "n_in  = ", data%n_in
write (u, "(1x,A,I0)")  "n_out = ", data%n_out
write (u, "(1x,A,I0)")  "n_flv = ", data%n_flv
write (u, "(1x,A,I0)")  "n_hel = ", data%n_hel
write (u, "(1x,A,I0)")  "n_col = ", data%n_col
write (u, "(1x,A,I0)")  "n_cin = ", data%n_cin
write (u, "(1x,A,I0)")  "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
do i = 2, 16
```

```
      write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
   end do
   write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
   write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
   write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
   write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

   write (u, "(A)")
   write (u, "(A)")  "* Set parameters for omega1_a and initialize:"
   write (u, "(A)")

   par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
   write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
   write (u, "(2x,A,F6.4)")  "me   = ", par(2)
   write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
   write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

   write (u, "(A)")
   write (u, "(A)")  "* Set kinematics:"
   write (u, "(A)")

   p = reshape ([ &
        1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
        1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
        1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
        1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
        ], [4,4])
   do i = 1, 4
       write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
   end do

   select type (driver)
   type is (omega_driver_t)
      call driver%init (par)

      call driver%new_event (p)

      write (u, "(A)")
      write (u, "(A)")  "* Compute matrix element:"
      write (u, "(A)")

      call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
      write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

      call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
   end select

   call lib%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: prc_omega_1"

end subroutine prc_omega_1
```

Check `prc_omega_t` wrapper and options

The process is $e^-e^+ \to e^-e^+$ for vanishing masses and $e = 0.3$. We build the library using the high-level procedure `omega_make_process_component` and the "black box" `prc_omega_t` object. Two variants with different settings for restrictions and OpenMP.

For `GNU make`, `makeflags` is set to `-j1`. This eliminates a potential clash with a `-j<n>` flag if this test is called from a parallel make.

⟨*Omega interface: execute tests*⟩+≡
```
    call test (prc_omega_2, "prc_omega_2", &
        "OMega option passing", &
        u, results)
```

⟨*Omega interface: tests*⟩+≡
```
    subroutine prc_omega_2 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      type(string_t) :: model_name
      type(model_list_t) :: model_list
      type(model_t), pointer :: model => null ()
      type(var_list_t), pointer :: var_list => null ()
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(string_t) :: restrictions
      type(process_component_def_t), pointer :: config
      type(prc_omega_t) :: prc1, prc2
      type(process_constants_t) :: data
      integer, parameter :: cdf = c_default_float
      integer, parameter :: ci = c_int
      real(cdf), dimension(:), allocatable :: par
      real(cdf), dimension(0:3,4) :: p
      complex(c_default_complex) :: amp
      integer :: i
      logical :: exist

      write (u, "(A)")  "* Test output: prc_omega_2"
      write (u, "(A)")  "*   Purpose: create simple processes with OMega"
      write (u, "(A)")  "*            use the prc_omega wrapper for this"
      write (u, "(A)")  "*            and check OMega options"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize a process library with two entries, &
          &different options."
      write (u, "(A)")  "* (1) e- e+ -> e- e+   &
          &(all diagrams, no OpenMP, report progress)"
      write (u, "(A)")  "* (2) e- e+ -> e- e+   &
          &(s-channel only, with OpenMP, report progress to file)"

      call lib%init (var_str ("omega2"))
      call os_data_init (os_data)
      call syntax_model_file_init ()
```

```
model_name = "QED"
call model_list%read_model &
     (var_str ("QED"), var_str ("QED.mdl"), os_data, model)
var_list => model_get_var_list_ptr (model)

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e-"), var_str ("e+")]
prt_out = prt_in
restrictions = "3+4~A"

allocate (entry)
call entry%init (var_str ("omega2_a"), &
     model, n_in = 2, n_components = 2)

call omega_make_process_component (entry, 1, &
     model_name, prt_in, prt_out, &
     report_progress=.true.)
call omega_make_process_component (entry, 2, &
     model_name, prt_in, prt_out, &
     restrictions=restrictions, openmp_support=.true., &
     extra_options=var_str ("-fusion:progress_file omega2.log"))

call lib%append (entry)

write (u, "(A)")
write (u, "(A)")  "* Remove left-over file"
write (u, "(A)")

call delete_file ("omega2.log")
inquire (file="omega2.log", exist=exist)
write (u, "(1x,A,L1)")  "omega2.log exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Build and load library"

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")
write (u, "(A)")  "* Check extra output of OMega"
write (u, "(A)")

inquire (file="omega2.log", exist=exist)
write (u, "(1x,A,L1)")  "omega2.log exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                   = ", &
```

```
      lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes             = ", &
      lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega2_a and initialize:"
write (u, "(A)")

call var_list_set_real (var_list, var_str ("ee"), 0.3_default, &
      is_known = .true.)
call var_list_set_real (var_list, var_str ("me"), 0._default, &
      is_known = .true.)
call var_list_set_real (var_list, var_str ("mmu"), 0._default, &
      is_known = .true.)
call var_list_set_real (var_list, var_str ("mtau"), 0._default, &
      is_known = .true.)
call model_parameters_to_c_array (model, par)

write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

call prc1%set_parameters (model)
call prc2%set_parameters (model)

write (u, "(A)")
write (u, "(A)")  "* Constants of omega2_a_i1:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega2_a"), 1)
call prc1%init (config%get_core_def_ptr (), &
      lib, var_str ("omega2_a"), 1)
call prc1%get_constants (data, 1)

write (u, "(1x,A,A)")  "component ID    = ", &
      char (data%id)
write (u, "(1x,A,L1)") "openmp supported = ", &
      data%openmp_supported
write (u, "(1x,A,A,A)") "model name       = '", &
      char (data%model_name), "'"

write (u, "(A)")
write (u, "(A)")  "* Constants of omega2_a_i2:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega2_a"), 2)
call prc2%init (config%get_core_def_ptr (), &
      lib, var_str ("omega2_a"), 2)
call prc2%get_constants (data, 1)

write (u, "(1x,A,A)")  "component ID    = ", &
      char (data%id)
write (u, "(1x,A,L1)") "openmp supported = ", &
```

```
      data%openmp_supported
write (u, "(1x,A,A,A)") "model name        = '", &
     char (data%model_name), "'"

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

select type (driver => prc1%driver)
type is (omega_driver_t)
   call driver%new_event (p)
   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
   write (u, "(2x,A,1x,E11.4)") "(1) |amp (1, 6, 1)| =", abs (amp)
end select

select type (driver => prc2%driver)
type is (omega_driver_t)
   call driver%new_event (p)
   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
   write (u, "(2x,A,1x,E11.4)") "(2) |amp (1, 6, 1)| =", abs (amp)
end select

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, sqrt(0.5_cdf), 0.0_cdf, sqrt(0.5_cdf), &
     1.0_cdf,-sqrt(0.5_cdf), 0.0_cdf,-sqrt(0.5_cdf) &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")
```

```
      select type (driver => prc1%driver)
      type is (omega_driver_t)
         call driver%new_event (p)
         call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
         write (u, "(2x,A,1x,E11.4)") "(1) |amp (1, 6, 1)| =", abs (amp)
      end select

      select type (driver => prc2%driver)
      type is (omega_driver_t)
         call driver%new_event (p)
         call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
         write (u, "(2x,A,1x,E11.4)") "(2) |amp (1, 6, 1)| =", abs (amp)
      end select

      call lib%final ()

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prc_omega_2"

    end subroutine prc_omega_2
```

Check helicity selection

The process is $e^- e^+ \to e^- e^+$ for vanishing masses. We call the matrix element
several times to verify the switching off of irrelevant helicities.

⟨*Omega interface: execute tests*⟩+≡
```
    call test (prc_omega_3, "prc_omega_3", &
         "helicity selection", &
         u, results)
```

⟨*Omega interface: tests*⟩+≡
```
    subroutine prc_omega_3 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      type(string_t) :: model_name
      type(model_list_t) :: model_list
      type(model_t), pointer :: model => null ()
      type(var_list_t), pointer :: var_list => null ()
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(process_component_def_t), pointer :: config
      type(prc_omega_t) :: prc1
      type(process_constants_t) :: data
      integer, parameter :: cdf = c_default_float
      real(cdf), dimension(:), allocatable :: par
      real(cdf), dimension(0:3,4) :: p
      type(helicity_selection_t) :: helicity_selection
      integer :: i, h
```

```
write (u, "(A)")  "* Test output: prc_omega_3"
write (u, "(A)")  "*   Purpose: create simple process with OMega"
write (u, "(A)")  "*            and check helicity selection"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library."
write (u, "(A)")  "* (1) e- e+ -> e- e+   (all diagrams, no OpenMP)"

call lib%init (var_str ("omega3"))
call os_data_init (os_data)
call syntax_model_file_init ()

model_name = "QED"
call model_list%read_model &
     (var_str ("QED"), var_str ("QED.mdl"), os_data, model)
var_list => model_get_var_list_ptr (model)

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e-"), var_str ("e+")]
prt_out = prt_in

allocate (entry)
call entry%init (var_str ("omega3_a"), &
     model, n_in = 2, n_components = 1)

call omega_make_process_component (entry, 1, &
     model_name, prt_in, prt_out)
call lib%append (entry)

write (u, "(A)")
write (u, "(A)")  "* Build and load library"

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes              = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega3_a and initialize:"
write (u, "(A)")

call var_list_set_real (var_list, var_str ("ee"), 0.3_default, &
     is_known = .true.)
call var_list_set_real (var_list, var_str ("me"), 0._default, &
```

```
      is_known = .true.)
call var_list_set_real (var_list, var_str ("mmu"), 0._default, &
      is_known = .true.)
call var_list_set_real (var_list, var_str ("mtau"), 0._default, &
      is_known = .true.)
call model_parameters_to_c_array (model, par)

write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

call prc1%set_parameters (model, helicity_selection)

write (u, "(A)")
write (u, "(A)")  "* Helicity states of omega3_a_i1:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega3_a"), 1)
call prc1%init (config%get_core_def_ptr (), &
      lib, var_str ("omega3_a"), 1)
call prc1%get_constants (data, 1)

do i = 1, data%n_hel
   write (u, "(3x,I2,':',4(1x,I2))") i, data%hel_state(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Initially allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
   write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

write (u, "(A)")
write (u, "(A)")  "* Reset helicity selection (cutoff = 4)"
write (u, "(A)")

helicity_selection%active = .true.
helicity_selection%threshold = 1e10_default
helicity_selection%cutoff = 4
call helicity_selection%write (u)

call prc1%set_parameters (model, helicity_selection)
call prc1%reset_helicity_selection ()

write (u, "(A)")
write (u, "(A)")  "* Allowed helicities:"
write (u, "(A)")
```

```
write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
   write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute scattering matrix 5 times"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]

select type (driver => prc1%driver)
type is (omega_driver_t)
   do i = 1, 5
      call driver%new_event (p)
      write (u, "(2x,I2)", advance = "no")  i
      do h = 1, data%n_hel
         write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
      end do
      write (u, "(A)")
   end do
end select

write (u, "(A)")
write (u, "(A)")  "* Reset helicity selection again"
write (u, "(A)")

call prc1%activate_parameters ()

write (u, "(A)")  "* Allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
   write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")
```

```
   call lib%final ()

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: prc_omega_3"

  end subroutine prc_omega_3
```

QCD coupling

The process is $u\bar{u} \to d\bar{d}$ for vanishing masses. We compute the amplitude for a fixed configuration once, then reset α_s, then compute again.

For `GNU make`, `makeflags` is set to `-j1`. This eliminates a potential clash with a `-j<n>` flag if this test is called from a parallel make.

⟨*Omega interface: execute tests*⟩+≡
```
   call test (prc_omega_4, "prc_omega_4", &
       "update QCD alpha", &
       u, results)
```

⟨*Omega interface: tests*⟩+≡
```
   subroutine prc_omega_4 (u)
     integer, intent(in) :: u
     type(process_library_t) :: lib
     class(prc_core_def_t), allocatable :: def
     type(process_def_entry_t), pointer :: entry
     type(os_data_t) :: os_data
     type(string_t) :: model_name
     type(string_t), dimension(:), allocatable :: prt_in, prt_out
     type(process_constants_t) :: data
     class(prc_core_driver_t), allocatable :: driver
     integer, parameter :: cdf = c_default_float
     integer, parameter :: ci = c_int
     real(cdf), dimension(6) :: par
     real(cdf), dimension(0:3,4) :: p
     logical(c_bool) :: flag
     complex(c_default_complex) :: amp
     integer :: i
     real(cdf) :: alpha_s

     write (u, "(A)")  "* Test output: prc_omega_4"
     write (u, "(A)")  "*   Purpose: create a QCD process with OMega"
     write (u, "(A)")  "*            and check alpha_s dependence"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize a process library with one entry"
     write (u, "(A)")
     call lib%init (var_str ("prc_omega_4_lib"))
     call os_data_init (os_data)

     model_name = "QCD"
```

2514

```
allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("u"), var_str ("ubar")]
prt_out = [var_str ("d"), var_str ("dbar")]

allocate (omega_def_t :: def)
select type (def)
type is (omega_def_t)
   call def%init (model_name, prt_in, prt_out)
end select
allocate (entry)
call entry%init (var_str ("prc_omega_4_p"), model_name = model_name, &
     n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("omega"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure and compile process"
write (u, "(A)")
call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters:"
write (u, "(A)")

alpha_s = 0.1178_cdf

par = [alpha_s, 0._cdf, 0._cdf, 0._cdf, 173.1_cdf, 1.523_cdf]
write (u, "(2x,A,F8.4)")  "alpha_s = ", par(1)
write (u, "(2x,A,F8.4)")  "ms      = ", par(2)
write (u, "(2x,A,F8.4)")  "mc      = ", par(3)
write (u, "(2x,A,F8.4)")  "mb      = ", par(4)
write (u, "(2x,A,F8.4)")  "mtop    = ", par(5)
write (u, "(2x,A,F8.4)")  "wtop    = ", par(6)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     100.0_cdf, 0.0_cdf, 0.0_cdf, 100.0_cdf, &
     100.0_cdf, 0.0_cdf, 0.0_cdf,-100.0_cdf, &
     100.0_cdf, 100.0_cdf, 0.0_cdf, 0.0_cdf, &
```

2515

```
             100.0_cdf,-100.0_cdf, 0.0_cdf, 0.0_cdf &
          ], [4,4])
      do i = 1, 4
         write (u, "(2x,A,I0,A,4(1x,F7.1))")  "p", i, " =", p(:,i)
      end do

      call lib%connect_process (var_str ("prc_omega_4_p"), 1, data, driver)

      select type (driver)
      type is (omega_driver_t)
         call driver%init (par)

         write (u, "(A)")
         write (u, "(A)")  "* Compute matrix element:"
         write (u, "(A)")

         call driver%new_event (p)

         call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
         write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

         call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
         write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

         write (u, "(A)")
         write (u, "(A)")  "* Double alpha_s and compute matrix element again:"
         write (u, "(A)")

         call driver%update_alpha_s (2 * alpha_s)
         call driver%new_event (p)

         call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
         write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

         call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
         write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
      end select

      call lib%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prc_omega_4"

   end subroutine prc_omega_4
```

Amplitude and QCD coupling

The same process as before. Here, we initialize with a running α_s coupling and compute twice with different scales. We use the high-level method `compute_amplitude`.

⟨*Omega interface: execute tests*⟩+≡
```
   call test (prc_omega_5, "prc_omega_5", &
        "running QCD alpha", &
```

```
            u, results)
```

⟨*Omega interface: tests*⟩+≡

```
    subroutine prc_omega_5 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      class(prc_core_def_t), allocatable :: def
      type(process_component_def_t), pointer :: cdef_ptr
      class(prc_core_def_t), pointer :: def_ptr
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(string_t) :: model_name
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(qcd_t) :: qcd
      class(prc_core_t), allocatable :: core
      class(workspace_t), allocatable :: tmp
      type(vector4_t), dimension(4) :: p
      complex(default) :: amp
      real(default) :: fac_scale
      integer :: i

      write (u, "(A)")  "* Test output: prc_omega_5"
      write (u, "(A)")  "*   Purpose: create a QCD process with OMega"
      write (u, "(A)")  "*           and check alpha_s dependence"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize a process library with one entry"
      write (u, "(A)")
      call lib%init (var_str ("prc_omega_5_lib"))
      call os_data_init (os_data)

      call syntax_model_file_init ()
      call model_list%read_model (var_str ("QCD"), var_str ("QCD.mdl"), &
           os_data, model)
      model_name = "QCD"

      allocate (prt_in (2), prt_out (2))
      prt_in = [var_str ("u"), var_str ("ubar")]
      prt_out = [var_str ("d"), var_str ("dbar")]

      allocate (omega_def_t :: def)
      select type (def)
      type is (omega_def_t)
         call def%init (model_name, prt_in, prt_out)
      end select
      allocate (entry)
      call entry%init (var_str ("prc_omega_5_p"), model_name = model_name, &
           n_in = 2, n_components = 1)
      call entry%import_component (1, n_out = size (prt_out), &
           prt_in  = new_prt_spec (prt_in), &
           prt_out = new_prt_spec (prt_out), &
           method  = var_str ("omega"), &
           variant = def)
```

```
                call lib%append (entry)

                write (u, "(A)")  "* Configure and compile process"
                write (u, "(A)")
                call lib%configure (os_data)
                call lib%write_makefile (os_data, force = .true.)
                call lib%clean (os_data, distclean = .false.)
                call lib%write_driver (force = .true.)
                call lib%load (os_data)

                write (u, "(A)")  "* Probe library API"
                write (u, "(A)")

                write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

                write (u, "(A)")
                write (u, "(A)")  "* Set kinematics"
                write (u, "(A)")

                p(1) = vector4_moving (100._default, 100._default, 3)
                p(2) = vector4_moving (100._default,-100._default, 3)
                p(3) = vector4_moving (100._default, 100._default, 1)
                p(4) = vector4_moving (100._default,-100._default, 1)
                do i = 1, 4
                   call vector4_write (p(i), u)
                end do

                write (u, "(A)")
                write (u, "(A)")  "* Setup QCD data"
                write (u, "(A)")

                allocate (alpha_qcd_from_scale_t :: qcd%alpha)

                write (u, "(A)")  "* Setup process core"
                write (u, "(A)")

                allocate (prc_omega_t :: core)
                cdef_ptr => lib%get_component_def_ptr (var_str ("prc_omega_5_p"), 1)
                def_ptr => cdef_ptr%get_core_def_ptr ()

                select type (core)
                type is (prc_omega_t)
                   call core%allocate_workspace (tmp)
                   call core%set_parameters (model, qcd = qcd)
                   call core%init (def_ptr, lib, var_str ("prc_omega_5_p"), 1)
                   call core%write (u)

                   write (u, "(A)")
                   write (u, "(A)")  "* Compute matrix element"
                   write (u, "(A)")

                   fac_scale = 100
                   write (u, "(1x,A,F4.0)")  "factorization scale = ", fac_scale
```

```
            amp = core%compute_amplitude &
                 (1, p, 1, 6, 1, fac_scale, 100._default)

            write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

            write (u, "(A)")
            write (u, "(A)")  "* Modify factorization scale and &
                 &compute matrix element again"
            write (u, "(A)")

            fac_scale = 200
            write (u, "(1x,A,F4.0)")  "factorization scale = ", fac_scale

            amp = core%compute_amplitude &
                 (1, p, 1, 6, 1, fac_scale, 100._default)

            write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

         end select

         call lib%final ()

         call model_list%final ()
         call syntax_model_file_final ()

         write (u, "(A)")
         write (u, "(A)")  "* Test output end: prc_omega_5"

    end subroutine prc_omega_5
```

Generate matrix element diagrams

The same process as before. No amplitude is computed here, instead we just generate Feynman (and color flow) diagrams, and check whether PS and PDF files have been generated. This test is only run if event analysis is possible.

⟨*Omega interface: execute diags tests*⟩≡
```
    call test (prc_omega_diags_1, "prc_omega_diags_1", &
         "generate Feynman diagrams", &
         u, results)
```

⟨*Omega interface: tests*⟩+≡
```
    subroutine prc_omega_diags_1 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      class(prc_core_def_t), allocatable :: def
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model
      type(string_t) :: model_name
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(string_t) :: diags_file, pdf_file, ps_file
      logical :: exist, exist_pdf, exist_ps
```

```
integer :: iostat, u_diags
character(128) :: buffer

write (u, "(A)")  "* Test output: prc_omega_diags_1"
write (u, "(A)")  "*   Purpose: generate Feynman diagrams"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library with one entry"
write (u, "(A)")
call lib%init (var_str ("prc_omega_diags_1_lib"))
call os_data_init (os_data)

call syntax_model_file_init ()
call model_list%read_model (var_str ("SM"), var_str ("SM.mdl"), &
     os_data, model)
model_name = "SM"

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("u"), var_str ("ubar")]
prt_out = [var_str ("d"), var_str ("dbar")]

allocate (omega_def_t :: def)
select type (def)
type is (omega_def_t)
   call def%init (model_name, prt_in, prt_out, &
        diags = .true., diags_color = .true.)
end select
allocate (entry)
call entry%init (var_str ("prc_omega_diags_1_p"), model_name = model_name, &
     n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("omega"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure and compile process"
write (u, "(A)")  "    and generate diagrams"
write (u, "(A)")
call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true., testflag = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")  "* Probe library API"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

write (u, "(A)")  "* Check produced diagram files"
write (u, "(A)")
```

```fortran
    diags_file = "prc_omega_diags_1_p_i1_diags.tex"
    ps_file  = "prc_omega_diags_1_p_i1_diags.ps"
    pdf_file = "prc_omega_diags_1_p_i1_diags.pdf"
    inquire (file = char (diags_file), exist = exist)
    if (exist) then
       u_diags = free_unit ()
       open (u_diags, file = char (diags_file), action = "read", status = "old")
       iostat = 0
       do while (iostat == 0)
          read (u_diags, "(A)", iostat = iostat)  buffer
          if (iostat == 0)  write (u, "(A)")  trim (buffer)
       end do
       close (u_diags)
    else
       write (u, "(A)")  "[Feynman diagrams LaTeX file is missing]"
    end if
    inquire (file = char (ps_file), exist = exist_ps)
    if (exist_ps) then
       write (u, "(A)")  "[Feynman diagrams Postscript file exists and is nonempty]"
    else
       write (u, "(A)")  "[Feynman diagrams Postscript file is missing/non-regular]"
    end if
    inquire (file = char (pdf_file), exist = exist_pdf)
    if (exist_pdf) then
       write (u, "(A)")  "[Feynman diagrams PDF file exists and is nonempty]"
    else
       write (u, "(A)")  "[Feynman diagrams PDF file is missing/non-regular]"
    end if

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"
    write (u, "(A)")

    call lib%final ()

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_omega_diags_1"

  end subroutine prc_omega_diags_1
```

Chapter 19

Event I/O

In this chapter we provide a generic handler for event I/O and the various implementations: raw, HepMC, Les Houches, etc.

19.1 Event Sample Data

We define a simple and transparent container for (meta)data that are associated with an event sample.

⟨eio_data.f90⟩≡
  ⟨*File header*⟩

  module eio_data

    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use diagnostics !NODEP!
    use unit_tests

    use events

  ⟨*EIO data: public*⟩

  ⟨*EIO data: types*⟩

  contains

  ⟨*EIO data: procedures*⟩

  ⟨*EIO data: tests*⟩

  end module eio_data

19.1.1 Event Sample Data

These are data that apply to an event sample as a whole. They are given in an easily portable form (no fancy structure) and are used for initializing event

formats.

There are two MD5 sums here. `md5sum_proc` depends only on the definition of the contributing processes. A sample with matching checksum can be rescanned with modified model parameters, beam structure etc, to recalculate observables. `md5sum_config` includes all relevant data. Rescanning a sample with matching checksum will produce identical observables. (A third checksum might be added which depends on the event sample itself. This is not needed, so far.)

If alternate weights are part of the event sample (`n_alt` nonzero), there is a configuration MD5 sum for each of them.

⟨*EIO data: public*⟩≡
```
    public :: event_sample_data_t
```

⟨*EIO data: types*⟩≡
```
    type :: event_sample_data_t
       character(32) :: md5sum_prc = ""
       character(32) :: md5sum_cfg = ""
       logical :: unweighted = .true.
       logical :: negative_weights = .false.
       integer :: norm_mode = NORM_UNDEFINED
       integer :: n_beam = 0
       integer, dimension(2) :: pdg_beam = 0
       real(default), dimension(2) :: energy_beam = 0
       integer :: n_proc = 0
       integer :: n_evt = 0
       integer :: split_n_evt = 0
       integer :: split_index = 0
       real(default) :: total_cross_section = 0
       integer, dimension(:), allocatable :: proc_num_id
       integer :: n_alt = 0
       character(32), dimension(:), allocatable :: md5sum_alt
       real(default), dimension(:), allocatable :: cross_section
       real(default), dimension(:), allocatable :: error
     contains
     ⟨EIO data: event sample data: TBP⟩
    end type event_sample_data_t
```

Initialize: allocate for the number of processes

⟨*EIO data: event sample data: TBP*⟩≡
```
    procedure :: init => event_sample_data_init
```

⟨*EIO data: procedures*⟩≡
```
    subroutine event_sample_data_init (data, n_proc, n_alt)
      class(event_sample_data_t), intent(out) :: data
      integer, intent(in) :: n_proc
      integer, intent(in), optional :: n_alt
      data%n_proc = n_proc
      allocate (data%proc_num_id (n_proc), source = 0)
      allocate (data%cross_section (n_proc), source = 0._default)
      allocate (data%error (n_proc), source = 0._default)
      if (present (n_alt)) then
         data%n_alt = n_alt
         allocate (data%md5sum_alt (n_alt))
```

```
        data%md5sum_alt = ""
      end if
    end subroutine event_sample_data_init
```

Output.
⟨*EIO data: event sample data: TBP*⟩+≡
```
    procedure :: write => event_sample_data_write
```

⟨*EIO data: procedures*⟩+≡
```
    subroutine event_sample_data_write (data, unit)
      class(event_sample_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "Event sample properties:"
      write (u, "(3x,A,A,A)")  "MD5 sum (proc)  = '", data%md5sum_prc, "'"
      write (u, "(3x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
      write (u, "(3x,A,L1)")  "unweighted       = ", data%unweighted
      write (u, "(3x,A,L1)")  "negative weights = ", data%negative_weights
      write (u, "(3x,A,A)")  "normalization    = ", &
           char (event_normalization_string (data%norm_mode))
      write (u, "(3x,A,I0)")  "number of beams  = ", data%n_beam
      write (u, "(5x,A,2(1x,I19))")  "PDG    = ", &
           data%pdg_beam(:data%n_beam)
      write (u, "(5x,A,2(1x,ES19.12))")  "Energy = ", &
           data%energy_beam(:data%n_beam)
      if (data%n_evt > 0) then
         write (u, "(3x,A,I0)")  "number of events = ", data%n_evt
      end if
      if (data%total_cross_section /= 0) then
         write (u, "(3x,A,ES19.12)")  "total cross sec. = ", &
              data%total_cross_section
      end if
      write (u, "(3x,A,I0)")  "num of processes = ", data%n_proc
      do i = 1, data%n_proc
         write (u, "(3x,A,I0)")  "Process #", data%proc_num_id (i)
         select case (data%n_beam)
         case (1)
            write (u, "(5x,A,ES19.12)")  "Width = ", data%cross_section(i)
         case (2)
            write (u, "(5x,A,ES19.12)")  "CSec  = ", data%cross_section(i)
         end select
         write (u, "(5x,A,ES19.12)")  "Error = ", data%error(i)
      end do
      if (data%n_alt > 0) then
         write (u, "(3x,A,I0)")  "num of alt wgt   = ", data%n_alt
         do i = 1, data%n_alt
            write (u, "(5x,A,A,A,1x,I0)")  "MD5 sum (cfg)  = '", &
                 data%md5sum_alt(i), "'", i
         end do
      end if
    end subroutine event_sample_data_write
```

19.1.2 Unit tests

⟨EIO data: public⟩+≡
```
    public :: eio_data_test
```

⟨EIO data: tests⟩≡
```
    subroutine eio_data_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨EIO data: execute tests⟩
    end subroutine eio_data_test
```

Event Sample Data

Print the contents of a sample data block.

⟨EIO data: execute tests⟩≡
```
    call test (eio_data_1, "eio_data_1", &
         "event sample data", &
         u, results)
```

⟨EIO data: tests⟩+≡
```
    subroutine eio_data_1 (u)
      integer, intent(in) :: u
      type(event_sample_data_t) :: data

      write (u, "(A)")  "* Test output: eio_data_1"
      write (u, "(A)")  "*   Purpose:  display event sample data"
      write (u, "(A)")

      write (u, "(A)")  "* Decay process, one component"
      write (u, "(A)")

      call data%init (1, 1)
      data%n_beam = 1
      data%pdg_beam(1) = 25
      data%energy_beam(1) = 125

      data%norm_mode = NORM_UNIT

      data%proc_num_id = [42]
      data%cross_section = [1.23e-4_default]
      data%error = 5e-6_default

      data%md5sum_prc = "abcdefghijklmnopabcdefghijklmnop"
      data%md5sum_cfg = "12345678901234561234567890123456"
      data%md5sum_alt(1) = "uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu"

      call data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Scattering process, two components"
      write (u, "(A)")

      call data%init (2)
```

```
      data%n_beam = 2
      data%pdg_beam = [2212, -2212]
      data%energy_beam = [8._default, 10._default]

      data%norm_mode = NORM_SIGMA

      data%proc_num_id = [12, 34]
      data%cross_section = [100._default, 88._default]
      data%error = [1._default, 0.1_default]

      call data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_data_1"

    end subroutine eio_data_1
```

Event Normalization

Check the functions for translating modes and updating weights.

⟨*EIO data: execute tests*⟩+≡
```
    call test (eio_data_2, "eio_data_2", &
         "event normalization", &
         u, results)
```

⟨*EIO data: tests*⟩+≡
```
    subroutine eio_data_2 (u)
      integer, intent(in) :: u
      type(string_t) :: s
      logical :: unweighted
      real(default) :: w, sigma
      integer :: n

      write (u, "(A)")  "* Test output: eio_data_2"
      write (u, "(A)")  "*   Purpose:  handle event normalization"
      write (u, "(A)")

      write (u, "(A)")  "* Normalization strings"
      write (u, "(A)")

      s = "auto"
      unweighted = .true.
      write (u, "(1x,A,1x,L1,1x,A)")  char (s), unweighted, &
           char (event_normalization_string &
           (event_normalization_mode (s, unweighted)))
      s = "AUTO"
      unweighted = .false.
      write (u, "(1x,A,1x,L1,1x,A)")  char (s), unweighted, &
           char (event_normalization_string &
           (event_normalization_mode (s, unweighted)))

      unweighted = .true.
```

2526

```
s = "1"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "1/n"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "Sigma"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "sigma/N"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))

write (u, "(A)")
write (u, "(A)")  "* Normalization update"
write (u, "(A)")

sigma = 5
n = 2

w0 = 1

w = w0
call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_S_N, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w

write (u, *)

w0 = 0.5

w = w0
call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_S_N, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w

write (u, *)
```

```
      w0 = 5.0

      w = w0
      call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_SIGMA)
      write (u, "(2(F6.3))")  w0, w
      w = w0
      call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_SIGMA)
      write (u, "(2(F6.3))")  w0, w
      w = w0
      call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_SIGMA)
      write (u, "(2(F6.3))")  w0, w
      w = w0
      call event_normalization_update (w, sigma, n, NORM_S_N, NORM_SIGMA)
      write (u, "(2(F6.3))")  w0, w

      write (u, *)

      w0 = 2.5

      w = w0
      call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_S_N)
      write (u, "(2(F6.3))")  w0, w
      w = w0
      call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_S_N)
      write (u, "(2(F6.3))")  w0, w
      w = w0
      call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_S_N)
      write (u, "(2(F6.3))")  w0, w
      w = w0
      call event_normalization_update (w, sigma, n, NORM_S_N, NORM_S_N)
      write (u, "(2(F6.3))")  w0, w

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_data_2"

    end subroutine eio_data_2
```

19.2   Abstract I/O Handler

This module defines an abstract object for event I/O and the associated methods.

There are `output` and `input` methods which write or read a single event from/to the I/O stream, respectively. The I/O stream itself may be a file, a common block, or an externally linked structure, depending on the concrete implementation.

A `write` method prints the current content of the implementation-dependent event record in human-readable form.

The `init_in`/`init_out` and `final` prepare and finalize the I/O stream, respectively. There is also a `switch_inout` method which turns an input stream into an output stream where events can be appended.

Optionally, output files can be split in chunks of well-defined size. The `split_out` method takes care of this.

⟨`eio_base.f90`⟩≡

  ⟨*File header*⟩

  `module eio_base`

    `use kinds !NODEP!`
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    `use diagnostics !NODEP!`
    `use unit_tests`

    `use lorentz !NODEP!`
    `use models`
    `use particles`
    `use beams`
    `use processes`
    `use events`
    `use eio_data`

  ⟨*Standard module head*⟩

  ⟨*EIO base: public*⟩

  ⟨*EIO base: types*⟩

  ⟨*EIO base: interfaces*⟩

  ⟨*EIO base: test types*⟩

  `contains`

  ⟨*EIO base: procedures*⟩

  ⟨*EIO base: tests*⟩

  `end module eio_base`

19.2.1 Process pointer

For handling multiple processes, we need an array of process pointers. This requires a wrapper:

⟨*EIO base: public*⟩≡
    `public :: process_ptr_t`

⟨*EIO base: types*⟩≡
    `type :: process_ptr_t`
      `type(process_t), pointer :: ptr => null ()`
    `end type process_ptr_t`

19.2.2 Type

We can assume that most implementations will need the file extension as a fixed string and, if they support file splitting, the current file index.

⟨*EIO base: public*⟩+≡
```
    public :: eio_t
```

⟨*EIO base: types*⟩+≡
```
    type, abstract :: eio_t
       type(string_t) :: sample
       type(string_t) :: extension
       type(string_t) :: filename
       logical :: split = .false.
       integer :: split_n_evt = 0
       integer :: split_index = 0
     contains
     ⟨EIO base: eio: TBP⟩
     end type eio_t
```

Write to screen. If possible, this should display the contents of the current event, i.e., the last one that was written or read.

⟨*EIO base: eio: TBP*⟩≡
```
    procedure (eio_write), deferred :: write
```

⟨*EIO base: interfaces*⟩≡
```
    abstract interface
       subroutine eio_write (object, unit)
         import
         class(eio_t), intent(in) :: object
         integer, intent(in), optional :: unit
       end subroutine eio_write
    end interface
```

Finalize. This should write/read footer data and close input/output channels.

⟨*EIO base: eio: TBP*⟩+≡
```
    procedure (eio_final), deferred :: final
```

⟨*EIO base: interfaces*⟩+≡
```
    abstract interface
       subroutine eio_final (object)
         import
         class(eio_t), intent(inout) :: object
       end subroutine eio_final
    end interface
```

Determine splitting parameters from the event sample data.

⟨*EIO base: eio: TBP*⟩+≡
```
    procedure :: set_splitting => eio_set_splitting
```

⟨*EIO base: procedures*⟩≡
```
    subroutine eio_set_splitting (eio, data)
      class(eio_t), intent(inout) :: eio
      type(event_sample_data_t), intent(in) :: data
      eio%split = data%split_n_evt > 0
```

```
         if (eio%split) then
            eio%split_n_evt = data%split_n_evt
            eio%split_index = data%split_index
         end if
      end subroutine eio_set_splitting
```

Generate a filename, taking a possible split index into account.

⟨*EIO base: eio: TBP*⟩+≡
```
      procedure :: set_filename => eio_set_filename
```

⟨*EIO base: procedures*⟩+≡
```
      subroutine eio_set_filename (eio)
        class(eio_t), intent(inout) :: eio
        character(32) :: buffer
        if (eio%split) then
           write (buffer, "(I0,'.')")  eio%split_index
           eio%filename = eio%sample // "." // trim (buffer) // eio%extension
        else
           eio%filename = eio%sample // "." // eio%extension
        end if
      end subroutine eio_set_filename
```

Initialize for output. We provide process pointers. This should open an event file if appropriate and write header data. Some methods may require event sample data.

⟨*EIO base: eio: TBP*⟩+≡
```
      procedure (eio_init_out), deferred :: init_out
```

⟨*EIO base: interfaces*⟩+≡
```
      abstract interface
         subroutine eio_init_out (eio, sample, process_ptr, data, success)
           import
           class(eio_t), intent(inout) :: eio
           type(string_t), intent(in) :: sample
           type(process_ptr_t), dimension(:), intent(in) :: process_ptr
           type(event_sample_data_t), intent(in), optional :: data
           logical, intent(out), optional :: success
         end subroutine eio_init_out
      end interface
```

Initialize for input. We provide process names. This should open an event file if appropriate and read header data. The `md5sum` can be used to check the integrity of the configuration, it it provides a checksum to compare with. In case the extension has changed the extension is also given as an argument.

The `data` argument is `intent(inout)`: we may read part of it and keep other parts and/or check them against the data in the file.

⟨*EIO base: eio: TBP*⟩+≡
```
      procedure (eio_init_in), deferred :: init_in
```

⟨*EIO base: interfaces*⟩+≡
```
      abstract interface
         subroutine eio_init_in (eio, sample, process_ptr, data, success, extension)
           import
```

2531

```
        class(eio_t), intent(inout) :: eio
        type(string_t), intent(in) :: sample
        type(string_t), intent(in), optional :: extension
        type(process_ptr_t), dimension(:), intent(in) :: process_ptr
        type(event_sample_data_t), intent(inout), optional :: data
        logical, intent(out), optional :: success
      end subroutine eio_init_in
    end interface
```

Re-initialize for output. This should change the status of any event file from
input to output and position it for appending new events.

⟨*EIO base: eio: TBP*⟩+≡
```
    procedure (eio_switch_inout), deferred :: switch_inout
```

⟨*EIO base: interfaces*⟩+≡
```
    abstract interface
      subroutine eio_switch_inout (eio, success)
        import
        class(eio_t), intent(inout) :: eio
        logical, intent(out), optional :: success
      end subroutine eio_switch_inout
    end interface
```

This is similar: split the output, i.e., close the current file and open a new
one. The default implementation does nothing. For the feature to work, an
implementation must override this.

⟨*EIO base: eio: TBP*⟩+≡
```
    procedure :: split_out => eio_split_out
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_split_out (eio)
      class(eio_t), intent(inout) :: eio
    end subroutine eio_split_out
```

Output an event. All data can be taken from the `event` record. The index `i_prc`
identifies the process among the processes that are contained in the current
sample. The `reading` flag, if present, indicates that the event was read from
file, not generated.

⟨*EIO base: eio: TBP*⟩+≡
```
    procedure (eio_output), deferred :: output
```

⟨*EIO base: interfaces*⟩+≡
```
    abstract interface
      subroutine eio_output (eio, event, i_prc, reading)
        import
        class(eio_t), intent(inout) :: eio
        type(event_t), intent(in), target :: event
        integer, intent(in) :: i_prc
        logical, intent(in), optional :: reading
      end subroutine eio_output
    end interface
```

Input an event. This should fill all event data that cannot be inferred from the associated process.

The input is broken down into two parts. First we read the `i_prc` index. So we know which process to expect in the subsequent event. If we have reached end of file, we also will know. Then, we read the event itself.

The parameter `iostat` is supposed to be set as the Fortran standard requires, negative for EOF and positive for error.

⟨*EIO base: eio: TBP*⟩+≡
```
    procedure (eio_input_i_prc), deferred :: input_i_prc
    procedure (eio_input_event), deferred :: input_event
```

⟨*EIO base: interfaces*⟩+≡
```
    abstract interface
       subroutine eio_input_i_prc (eio, i_prc, iostat)
         import
         class(eio_t), intent(inout) :: eio
         integer, intent(out) :: i_prc
         integer, intent(out) :: iostat
       end subroutine eio_input_i_prc
    end interface

    abstract interface
       subroutine eio_input_event (eio, event, iostat)
         import
         class(eio_t), intent(inout) :: eio
         type(event_t), intent(inout), target :: event
         integer, intent(out) :: iostat
       end subroutine eio_input_event
    end interface
```

19.2.3 Unit tests

⟨*EIO base: public*⟩+≡
```
    public :: eio_base_test
```

⟨*EIO base: tests*⟩≡
```
    subroutine eio_base_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨EIO base: execute tests⟩
    end subroutine eio_base_test
```

Test type for event I/O

The contents simulate the contents of an external file. We have the `sample` string as the file name, `sqrts` and the array of `process_id`s as metadata, and the array of momenta `event_p` as the list of events. The second index is the event index. The `event_i` component is the pointer to the current event, `event_n` is the total number of stored events.

⟨*EIO base: test types*⟩≡
```
    type, extends (eio_t) :: eio_test_t
```

2533

```
      real(default) :: sqrts
      type(string_t) :: process_id
      integer :: event_n = 0
      integer :: event_i = 0
      integer :: i_prc = 0
      type(vector4_t), dimension(:,:), allocatable :: event_p
    contains
    ⟨EIO base: eio test: TBP⟩
  end type eio_test_t
```

Write to screen. Pretend that this is an actual event format.

⟨*EIO base: eio test: TBP*⟩≡
```
    procedure :: write => eio_test_write
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_write (object, unit)
      class(eio_test_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "Test event stream"
      write (u, "(3x,A,A,A)")  "Process ID  = '", char (object%process_id), "'"
      write (u, "(3x,A,ES19.12)")  "sqrts       = ", object%sqrts
      if (object%event_i /= 0) then
          write (u, "(1x,A,I0,A)")  "Event #", object%event_i, ":"
          do i = 1, size (object%event_p, 1)
             call vector4_write (object%event_p(i, object%event_i), u)
          end do
      end if
    end subroutine eio_test_write
```

Finalizer. For the test case, we just reset the event count, but keep the stored "events". For the real implementations, the events would be stored on an external medium, so we would delete the object contents.

⟨*EIO base: eio test: TBP*⟩+≡
```
    procedure :: final => eio_test_final
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_final (object)
      class(eio_test_t), intent(inout) :: object
      object%event_i = 0
    end subroutine eio_test_final
```

Initialization: We store the process IDs and the energy from the beam-data object. We also allocate the momenta (i.e., the simulated event record) for a fixed maximum size of 10 events, 2 momenta each. There is only a single process.

⟨*EIO base: eio test: TBP*⟩+≡
```
    procedure :: init_out => eio_test_init_out
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_init_out (eio, sample, process_ptr, data, success)
      class(eio_test_t), intent(inout) :: eio
```

```
    type(string_t), intent(in) :: sample
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    type(beam_data_t), pointer :: beam_data
    eio%sample = sample
    eio%process_id = process_ptr(1)%ptr%get_id ()
    beam_data => process_ptr(1)%ptr%get_beam_data_ptr ()
    eio%sqrts = beam_data_get_sqrts (beam_data)
    eio%event_n = 0
    eio%event_i = 0
    allocate (eio%event_p (2, 10))
    if (present (success))  success = .true.
  end subroutine eio_test_init_out
```

Initialization for input. Nothing to do for the test type.

⟨*EIO base: eio test: TBP*⟩+≡
```
    procedure :: init_in => eio_test_init_in
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_init_in &
        (eio, sample, process_ptr, data, success, extension)
      class(eio_test_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      if (present (success))  success = .true.
    end subroutine eio_test_init_in
```

Switch from output to input. Again, nothing to do for the test type.

⟨*EIO base: eio test: TBP*⟩+≡
```
    procedure :: switch_inout => eio_test_switch_inout
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_switch_inout (eio, success)
      class(eio_test_t), intent(inout) :: eio
      logical, intent(out), optional :: success
      if (present (success))  success = .true.
    end subroutine eio_test_switch_inout
```

Output. Increment the event counter and store the momenta of the current event.

⟨*EIO base: eio test: TBP*⟩+≡
```
    procedure :: output => eio_test_output
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_output (eio, event, i_prc, reading)
      class(eio_test_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      logical, intent(in), optional :: reading
      integer, intent(in) :: i_prc
```

```
      type(particle_set_t), pointer :: pset
      type(particle_t) :: prt
      eio%event_n = eio%event_n + 1
      eio%event_i = eio%event_n
      eio%i_prc = i_prc
      pset => event%get_particle_set_ptr ()
      prt = particle_set_get_particle (pset, 3)
      eio%event_p(1, eio%event_i) = particle_get_momentum (prt)
      prt = particle_set_get_particle (pset, 4)
      eio%event_p(2, eio%event_i) = particle_get_momentum (prt)
    end subroutine eio_test_output
```

Input. Increment the event counter and retrieve the momenta of the current event. For the test case, we do not actually modify the current event.

⟨*EIO base: eio test: TBP*⟩+≡
```
    procedure :: input_i_prc => eio_test_input_i_prc
    procedure :: input_event => eio_test_input_event
```

⟨*EIO base: procedures*⟩+≡
```
    subroutine eio_test_input_i_prc (eio, i_prc, iostat)
      class(eio_test_t), intent(inout) :: eio
      integer, intent(out) :: i_prc
      integer, intent(out) :: iostat
      i_prc = eio%i_prc
      iostat = 0
    end subroutine eio_test_input_i_prc

    subroutine eio_test_input_event (eio, event, iostat)
      class(eio_test_t), intent(inout) :: eio
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      eio%event_i = eio%event_i + 1
      iostat = 0
    end subroutine eio_test_input_event
```

Test I/O methods

We test the implementation of all I/O methods using the `eio_test_t` type.

⟨*EIO base: execute tests*⟩≡
```
    call test (eio_base_1, "eio_base_1", &
        "read and write event contents", &
        u, results)
```

⟨*EIO base: tests*⟩+≡
```
    subroutine eio_base_1 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      class(eio_t), allocatable :: eio
      integer :: i_prc,  iostat
```

```
      type(string_t) :: sample

      write (u, "(A)")  "* Test output: eio_base_1"
      write (u, "(A)")  "*   Purpose: generate and read/write an event"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
      process_ptr%ptr => process
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_test1"

      allocate (eio_test_t :: eio)

      call eio%init_out (sample, [process_ptr])
      call event%generate (1, [0._default, 0._default])
      call eio%output (event, 42)
      call eio%write (u)
      call eio%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Re-read the event"
      write (u, "(A)")

      call eio%init_in (sample, [process_ptr])
      call eio%input_i_prc (i_prc, iostat)
      call eio%input_event (event, iostat)
      call eio%write (u)
      write (u, "(A)")
      write (u, "(1x,A,I0)")  "i = ", i_prc

      write (u, "(A)")
      write (u, "(A)")  "* Generate and append another event"
      write (u, "(A)")

      call eio%switch_inout ()
      call event%generate (1, [0._default, 0._default])
      call eio%output (event, 5)
      call eio%write (u)
      call eio%final ()
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Re-read both events"
      write (u, "(A)")

      call eio%init_in (sample, [process_ptr])
      call eio%input_i_prc (i_prc, iostat)
      call eio%input_event (event, iostat)
      call eio%input_i_prc (i_prc, iostat)
      call eio%input_event (event, iostat)
      call eio%write (u)
      write (u, "(A)")
      write (u, "(1x,A,I0)")   "i = ", i_prc

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call eio%final ()
      deallocate (eio)

      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: eio_base_1"

    end subroutine eio_base_1
```

19.3   Raw Event I/O

The raw format is for internal use only. All data are stored unformatted, so
they can be efficiently be re-read on the same machine, but not necessarily on
another machine.

⟨eio_raw.f90⟩≡
  ⟨*File header*⟩

  module eio_raw

    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use diagnostics !NODEP!
    use unit_tests
```

```
      use lorentz !NODEP!
      use variables
      use models
      use particles
      use beams
      use processes
      use events
      use eio_data
      use eio_base
```

⟨*Standard module head*⟩

⟨*EIO raw: public*⟩

⟨*EIO raw: types*⟩

```
  contains
```

⟨*EIO raw: procedures*⟩

⟨*EIO raw: tests*⟩

```
  end module eio_raw
```

19.3.1   Type

⟨*EIO raw: public*⟩≡
```
    public :: eio_raw_t
```

⟨*EIO raw: types*⟩≡
```
    type, extends (eio_t) :: eio_raw_t
       logical :: reading = .false.
       logical :: writing = .false.
       integer :: unit = 0
       integer :: norm_mode = NORM_UNDEFINED
       real(default) :: sigma = 1
       integer :: n = 1
       integer :: n_alt = 0
       logical :: check = .false.
     contains
     ⟨EIO raw: eio raw: TBP⟩
    end type eio_raw_t
```

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO raw: eio raw: TBP*⟩≡
```
    procedure :: write => eio_raw_write
```

⟨*EIO raw: procedures*⟩≡
```
    subroutine eio_raw_write (object, unit)
      class(eio_raw_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
```

```
    write (u, "(1x,A)")  "Raw event stream:"
    write (u, "(3x,A,L1)")  "Check MD5 sum     = ", object%check
    if (object%n_alt > 0) then
       write (u, "(3x,A,I0)")  "Alternate weights = ", object%n_alt
    end if
    if (object%reading) then
       write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
    else if (object%writing) then
       write (u, "(3x,A,A)")  "Writing to file  = ", char (object%filename)
    else
       write (u, "(3x,A)")  "[closed]"
    end if
  end subroutine eio_raw_write
```

Finalizer: close any open file.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
    procedure :: final => eio_raw_final
```

⟨*EIO raw: procedures*⟩+≡
```
    subroutine eio_raw_final (object)
      class(eio_raw_t), intent(inout) :: object
      if (object%reading .or. object%writing) then
         write (msg_buffer, "(A,A,A)")  "Events: closing raw file '", &
              char (object%filename), "'"
         call msg_message ()
         close (object%unit)
         object%reading = .false.
         object%writing = .false.
      end if
    end subroutine eio_raw_final
```

Set the **check** flag which determines whether we compare checksums on input.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
    procedure :: set_parameters => eio_raw_set_parameters
```

⟨*EIO raw: procedures*⟩+≡
```
    subroutine eio_raw_set_parameters (eio, check, extension)
      class(eio_raw_t), intent(inout) :: eio
      logical, intent(in) :: check
      type(string_t), intent(in), optional :: extension
      eio%check = check
      if (present (extension)) then
         eio%extension = extension
      else
         eio%extension = "evx"
      end if
    end subroutine eio_raw_set_parameters
```

Initialize event writing.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
    procedure :: init_out => eio_raw_init_out
```

⟨*EIO raw: procedures*⟩+≡
```
subroutine eio_raw_init_out (eio, sample, process_ptr, data, success)
  class(eio_raw_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(process_ptr_t), dimension(:), intent(in) :: process_ptr
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  character(32) :: md5sum_prc, md5sum_cfg
  character(32), dimension(:), allocatable :: md5sum_alt
  integer :: i
  eio%extension = "evx"
  eio%filename = sample // "." // eio%extension
  eio%unit = free_unit ()
  write (msg_buffer, "(A,A,A)")  "Events: writing to raw file '", &
       char (eio%filename), "'"
  call msg_message ()
  eio%writing = .true.
  if (present (data)) then
     md5sum_prc = data%md5sum_prc
     md5sum_cfg = data%md5sum_cfg
     eio%norm_mode = data%norm_mode
     eio%sigma = data%total_cross_section
     eio%n = data%n_evt
     eio%n_alt = data%n_alt
     if (eio%n_alt > 0) then
        allocate (md5sum_alt (data%n_alt), source = data%md5sum_alt)
     end if
  else
     md5sum_prc = ""
     md5sum_cfg = ""
  end if
  open (eio%unit, file = char (eio%filename), form = "unformatted", &
       action = "write", status = "replace")
  write (eio%unit)  md5sum_prc
  write (eio%unit)  md5sum_cfg
  write (eio%unit)  eio%norm_mode
  write (eio%unit)  eio%n_alt
  do i = 1, eio%n_alt
     write (eio%unit)  md5sum_alt(i)
  end do
  if (present (success))  success = .true.
end subroutine eio_raw_init_out
```

Initialize event reading.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
procedure :: init_in => eio_raw_init_in
```

⟨*EIO raw: procedures*⟩+≡
```
subroutine eio_raw_init_in &
     (eio, sample, process_ptr, data, success, extension)
  class(eio_raw_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(process_ptr_t), dimension(:), intent(in) :: process_ptr
```

```fortran
type(event_sample_data_t), intent(inout), optional :: data
logical, intent(out), optional :: success
character(32) :: md5sum_prc, md5sum_cfg
character(32), dimension(:), allocatable :: md5sum_alt
integer :: i
if (present (success))  success = .true.
if (present (extension)) then
   eio%extension = extension
else
   eio%extension = "evx"
end if
eio%filename = sample // "." // eio%extension
eio%unit = free_unit ()
if (present (data)) then
   eio%sigma = data%total_cross_section
   eio%n = data%n_evt
end if
write (msg_buffer, "(A,A,A)")  "Events: reading from raw file '", &
     char (eio%filename), "'"
call msg_message ()
eio%reading = .true.
open (eio%unit, file = char (eio%filename), form = "unformatted", &
     action = "read", status = "old")
read (eio%unit)  md5sum_prc
read (eio%unit)  md5sum_cfg
read (eio%unit)  eio%norm_mode
read (eio%unit)  eio%n_alt
if (present (data)) then
   if (eio%n_alt /= data%n_alt) then
      if (present (success))  success = .false. !
      return
   end if
end if
allocate (md5sum_alt (eio%n_alt))
do i = 1, eio%n_alt
   read (eio%unit)  md5sum_alt(i)
end do
if (present (success)) then
   if (present (data)) then
      if (eio%check) then
         if (data%md5sum_prc /= "") then
            success = success .and. md5sum_prc == data%md5sum_prc
         end if
         if (data%md5sum_cfg /= "") then
            success = success .and. md5sum_cfg == data%md5sum_cfg
         end if
         do i = 1, eio%n_alt
            if (data%md5sum_alt(i) /= "") then
               success = success .and. md5sum_alt(i) == data%md5sum_alt(i)
            end if
         end do
      else
         call msg_warning ("Reading event file: MD5 sum check disabled")
      end if
```

```
      end if
    end if
  end subroutine eio_raw_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
    procedure :: switch_inout => eio_raw_switch_inout
```

⟨*EIO raw: procedures*⟩+≡
```
    subroutine eio_raw_switch_inout (eio, success)
      class(eio_raw_t), intent(inout) :: eio
      logical, intent(out), optional :: success
      write (msg_buffer, "(A,A,A)")  "Events: appending to raw file '", &
           char (eio%filename), "'"
      call msg_message ()
      close (eio%unit, status = "keep")
      eio%reading = .false.
      open (eio%unit, file = char (eio%filename), form = "unformatted", &
           action = "write", position = "append", status = "old")
      eio%writing = .true.
      if (present (success))  success = .true.
    end subroutine eio_raw_switch_inout
```

Output an event. Write first the event indices, then weight and squared matrix
element, then the particle set.

We always write the particle set of the hard process. (Note: this should be
reconsidered.) We do make a physical copy.

On output, we write the `prc` values for weight and sqme, since these are
the values just computed. On input, we store the values as `ref` values. The
caller can then decide whether to recompute values and thus obtain distinct `prc`
values, or just accept them.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
    procedure :: output => eio_raw_output
```

⟨*EIO raw: procedures*⟩+≡
```
    subroutine eio_raw_output (eio, event, i_prc, reading)
      class(eio_raw_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      logical, intent(in), optional :: reading
      integer, intent(in) :: i_prc
      type(particle_set_t) :: pset
      integer :: i
      if (eio%writing) then
         if (event%has_particle_set ()) then
            write (eio%unit)  i_prc
            write (eio%unit)  event%get_i_mci ()
            write (eio%unit)  event%get_i_term ()
            write (eio%unit)  event%get_channel ()
            write (eio%unit)  event%expr%weight_prc
            write (eio%unit)  event%expr%excess_prc
            write (eio%unit)  event%expr%sqme_prc
            do i = 1, eio%n_alt
               write (eio%unit)  event%expr%weight_alt(i)
```

2543

```
        write (eio%unit)  event%expr%sqme_alt(i)
     end do
     call event%get_particle_set_hard_proc (pset)
     call particle_set_write_raw (pset, eio%unit)
     call particle_set_final (pset)
  else
     call msg_bug ("Event: write raw: particle set is undefined")
  end if
else
   call eio%write ()
   call msg_fatal ("Raw event file is not open for writing")
end if
end subroutine eio_raw_output
```

Input an event.

Note: the particle set is physically copied. If there is a performance issue, we might choose to pointer-assign it instead, with a different version of `event%set_particle_set_hard_proc`.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
procedure :: input_i_prc => eio_raw_input_i_prc
procedure :: input_event => eio_raw_input_event
```

⟨*EIO raw: procedures*⟩+≡
```
subroutine eio_raw_input_i_prc (eio, i_prc, iostat)
  class(eio_raw_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  if (eio%reading) then
     read (eio%unit, iostat = iostat)  i_prc
  else
     call eio%write ()
     call msg_fatal ("Raw event file is not open for reading")
  end if
end subroutine eio_raw_input_i_prc

subroutine eio_raw_input_event (eio, event, iostat)
  class(eio_raw_t), intent(inout) :: eio
  type(event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  integer :: i_mci, i_term, channel, i
  real(default) :: weight, excess, sqme
  real(default), dimension(:), allocatable :: weight_alt, sqme_alt
  type(particle_set_t) :: pset
  if (eio%reading) then
     read (eio%unit, iostat = iostat)  i_mci
     if (iostat /= 0)  return
     read (eio%unit, iostat = iostat)  i_term
     if (iostat /= 0)  return
     read (eio%unit, iostat = iostat)  channel
     if (iostat /= 0)  return
     read (eio%unit, iostat = iostat)  weight
     if (iostat /= 0)  return
     read (eio%unit, iostat = iostat)  excess
     if (iostat /= 0)  return
```

```
            read (eio%unit, iostat = iostat)  sqme
            if (iostat /= 0)  return
            call event%reset ()
            call event%select (i_mci, i_term, channel)
            if (eio%norm_mode /= NORM_UNDEFINED) then
               call event_normalization_update (weight, &
                    eio%sigma, eio%n, event%get_norm_mode (), eio%norm_mode)
               call event_normalization_update (excess, &
                    eio%sigma, eio%n, event%get_norm_mode (), eio%norm_mode)
            end if
            call event%set (sqme_ref = sqme, weight_ref = weight, &
               excess_prc = excess)
            if (eio%n_alt /= 0) then
               allocate (sqme_alt (eio%n_alt), weight_alt (eio%n_alt))
               do i = 1, eio%n_alt
                  read (eio%unit, iostat = iostat)  weight_alt(i)
                  if (iostat /= 0)  return
                  read (eio%unit, iostat = iostat)  sqme_alt(i)
                  if (iostat /= 0)  return
               end do
               call event%set (sqme_alt = sqme_alt, weight_alt = weight_alt)
            end if
            call particle_set_read_raw (pset, eio%unit, iostat)
            if (iostat /= 0)  return
            if (associated (event%process)) then
               call particle_set_set_model (pset, event%process%get_model_ptr ())
            end if
            call event%set_particle_set_hard_proc (pset)
            call particle_set_final (pset)
         else
            call eio%write ()
            call msg_fatal ("Raw event file is not open for reading")
         end if
      end subroutine eio_raw_input_event
```

19.3.2 Unit tests

⟨*EIO raw: public*⟩+≡
```
    public :: eio_raw_test
```

⟨*EIO raw: tests*⟩≡
```
    subroutine eio_raw_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨EIO raw: execute tests⟩
    end subroutine eio_raw_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO raw: execute tests*⟩≡

```
      call test (eio_raw_1, "eio_raw_1", &
           "read and write event contents", &
           u, results)
```
⟨*EIO raw: tests*⟩+≡
```
     subroutine eio_raw_1 (u)
       integer, intent(in) :: u
       type(model_list_t) :: model_list
       type(event_t), allocatable, target :: event
       type(process_t), allocatable, target :: process
       type(process_ptr_t) :: process_ptr
       type(process_instance_t), allocatable, target :: process_instance
       class(eio_t), allocatable :: eio
       integer :: i_prc, iostat
       type(string_t) :: sample

       write (u, "(A)")  "* Test output: eio_raw_1"
       write (u, "(A)")  "*   Purpose: generate and read/write an event"
       write (u, "(A)")

       call syntax_model_file_init ()

       write (u, "(A)")  "* Initialize test process"

       allocate (process)
       process_ptr%ptr => process
       allocate (process_instance)
       call prepare_test_process (process, process_instance, model_list)
       call process_instance%setup_event_data ()

       allocate (event)
       call event%basic_init ()
       call event%connect (process_instance, process%get_model_ptr ())

       write (u, "(A)")
       write (u, "(A)")  "* Generate and write an event"
       write (u, "(A)")

       sample = "eio_raw_1"

       allocate (eio_raw_t :: eio)

       call eio%init_out (sample, [process_ptr])
       call event%generate (1, [0._default, 0._default])
       call event%evaluate_expressions ()
       call event%write (u)
       write (u, "(A)")

       call eio%output (event, i_prc = 42)
       call eio%write (u)
       call eio%final ()

       call event%final ()
       deallocate (event)
       call process_instance%final ()
```

2546

```
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event"
write (u, "(A)")

call eio%init_in (sample, [process_ptr])

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event):", iostat
call eio%write (u)

write (u, "(A)")
write (u, "(1x,A,I0)")  "i_prc = ", i_prc
write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate and append another event"
write (u, "(A)")

call eio%switch_inout ()
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%write (u)
write (u, "(A)")

call eio%output (event, i_prc = 5)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read both events"
write (u, "(A)")

call eio%init_in (sample, [process_ptr])

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
```

```
      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      call eio%input_i_prc (i_prc, iostat)
      if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc/1):", iostat
      call eio%input_event (event, iostat)
      if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event/1):", iostat
      call eio%input_i_prc (i_prc, iostat)
      if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc/2):", iostat
      call eio%input_event (event, iostat)
      if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event/2):", iostat
      call eio%write (u)

      write (u, "(A)")
      write (u, "(1x,A,I0)")  "i_prc = ", i_prc
      write (u, "(A)")
      call event%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call eio%final ()
      deallocate (eio)

      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_raw_1"

    end subroutine eio_raw_1
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO raw: execute tests*⟩+≡
```
    call test (eio_raw_2, "eio_raw_2", &
         "handle multiple weights", &
         u, results)
```

⟨*EIO raw: tests*⟩+≡
```
    subroutine eio_raw_2 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(var_list_t) :: var_list
```

```
type(event_t), allocatable, target :: event
type(process_t), allocatable, target :: process
type(process_ptr_t) :: process_ptr
type(process_instance_t), allocatable, target :: process_instance
type(event_sample_data_t) :: data
class(eio_t), allocatable :: eio
integer :: i_prc, iostat
type(string_t) :: sample

write (u, "(A)")  "* Test output: eio_raw_2"
write (u, "(A)")  "*   Purpose: generate and read/write an event"
write (u, "(A)")  "*            with multiple weights"
write (u, "(A)")

call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

call data%init (n_proc = 1, n_alt = 2)

call var_list_append_log (var_list, var_str ("?unweighted"), .false., &
     intrinsic = .true.)
call var_list_append_string (var_list, var_str ("$sample_normalization"), &
     var_str ("auto"), intrinsic = .true.)
call var_list_append_real (var_list, var_str ("safety_factor"), &
     1._default, intrinsic = .true.)

allocate (event)
call event%basic_init (var_list, n_alt = 2)
call event%connect (process_instance, process%get_model_ptr ())

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_raw_2"

allocate (eio_raw_t :: eio)

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%set (sqme_alt = [2._default, 3._default])
call event%set (weight_alt = [2 * event%weight_ref, 3 * event%weight_ref])
call event%store_alt_values ()
call event%check ()

call event%write (u)
```

```
write (u, "(A)")

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event"
write (u, "(A)")

call eio%init_in (sample, [process_ptr], data)

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
allocate (event)
call event%basic_init (var_list, n_alt = 2)
call event%connect (process_instance, process%get_model_ptr ())

call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event):", iostat
call eio%write (u)

write (u, "(A)")
write (u, "(1x,A,I0)")  "i_prc = ", i_prc
write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
deallocate (eio)

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

call model_list%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_raw_2"
```

```
        end subroutine eio_raw_2
```

19.4   Event Generation Checkpoints

This is an output-only format. Its only use is to write screen messages every n events, to inform the user about progress.

⟨eio_checkpoints.f90⟩≡

  ⟨*File header*⟩

```
  module eio_checkpoints

    use kinds !NODEP!
```
⟨*Use file utils*⟩
⟨*Use strings*⟩
```
    use diagnostics !NODEP!
    use unit_tests

    use cputime
    use models
    use processes
    use events
    use eio_data
    use eio_base
```

  ⟨*Standard module head*⟩

  ⟨*EIO checkpoints: public*⟩

  ⟨*EIO checkpoints: parameters*⟩

  ⟨*EIO checkpoints: types*⟩

```
  contains
```

  ⟨*EIO checkpoints: procedures*⟩

  ⟨*EIO checkpoints: tests*⟩

```
  end module eio_checkpoints
```

19.4.1   Type

⟨*EIO checkpoints: public*⟩≡
```
    public :: eio_checkpoints_t
```
⟨*EIO checkpoints: types*⟩≡
```
    type, extends (eio_t) :: eio_checkpoints_t
       logical :: active = .false.
       logical :: running = .false.
       integer :: val = 0
       integer :: n_events = 0
       integer :: n_read = 0
```

```
      integer :: i_evt = 0
      logical :: blank = .false.
      type(timer_t) :: timer
    contains
    ⟨EIO checkpoints: eio checkpoints: TBP⟩
    end type eio_checkpoints_t
```

19.4.2   Specific Methods

Set parameters that are specifically used with LHEF.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩≡
```
    procedure :: set_parameters => eio_checkpoints_set_parameters
```

⟨*EIO checkpoints: procedures*⟩≡
```
    subroutine eio_checkpoints_set_parameters (eio, checkpoint, blank)
      class(eio_checkpoints_t), intent(inout) :: eio
      integer, intent(in) :: checkpoint
      logical, intent(in), optional :: blank
      eio%val = checkpoint
      if (present (blank))  eio%blank = blank
    end subroutine eio_checkpoints_set_parameters
```

19.4.3   Common Methods

Output. This is not the actual event format, but a readable account of the current status.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
    procedure :: write => eio_checkpoints_write
```

⟨*EIO checkpoints: procedures*⟩+≡
```
    subroutine eio_checkpoints_write (object, unit)
      class(eio_checkpoints_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      if (object%active) then
        write (u, "(1x,A)")  "Event-sample checkpoints:  active"
        write (u, "(3x,A,I0)")  "interval  = ", object%val
        write (u, "(3x,A,I0)")  "n_events  = ", object%n_events
        write (u, "(3x,A,I0)")  "n_read    = ", object%n_read
        write (u, "(3x,A,I0)")  "n_current = ", object%i_evt
        write (u, "(3x,A,L1)")  "blanking  = ", object%blank
        call object%timer%write (u)
      else
        write (u, "(1x,A)")  "Event-sample checkpoints:  off"
      end if
    end subroutine eio_checkpoints_write
```

Finalizer: trivial.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
    procedure :: final => eio_checkpoints_final
```

⟨*EIO checkpoints: procedures*⟩+≡
```
    subroutine eio_checkpoints_final (object)
      class(eio_checkpoints_t), intent(inout) :: object
      object%active = .false.
    end subroutine eio_checkpoints_final
```

Activate checkpointing for event generation or writing.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
    procedure :: init_out => eio_checkpoints_init_out
```

⟨*EIO checkpoints: procedures*⟩+≡
```
    subroutine eio_checkpoints_init_out &
        (eio, sample, process_ptr, data, success)
      class(eio_checkpoints_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(in), optional :: data
      logical, intent(out), optional :: success
      if (present (data)) then
         if (eio%val > 0) then
            eio%active = .true.
            eio%i_evt = 0
            eio%n_read = 0
            eio%n_events = data%n_evt
         end if
      end if
      if (present (success))  success = .true.
    end subroutine eio_checkpoints_init_out
```

No checkpointing for event reading.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
    procedure :: init_in => eio_checkpoints_init_in
```

⟨*EIO checkpoints: procedures*⟩+≡
```
    subroutine eio_checkpoints_init_in &
        (eio, sample, process_ptr, data, success, extension)
      class(eio_checkpoints_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      call msg_bug ("Event checkpoints: event input not supported")
      if (present (success))  success = .false.
    end subroutine eio_checkpoints_init_in
```

Switch from input to output: also not supported.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
    procedure :: switch_inout => eio_checkpoints_switch_inout
```

⟨*EIO checkpoints: procedures*⟩+≡
```
    subroutine eio_checkpoints_switch_inout (eio, success)
      class(eio_checkpoints_t), intent(inout) :: eio
      logical, intent(out), optional :: success
```

```
      call msg_bug ("Event checkpoints: in-out switch not supported")
      if (present (success))  success = .false.
   end subroutine eio_checkpoints_switch_inout
```

Checkpoints: display progress for the current event, if applicable.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
   procedure :: output => eio_checkpoints_output
```

⟨*EIO checkpoints: procedures*⟩+≡
```
   subroutine eio_checkpoints_output (eio, event, i_prc, reading)
     class(eio_checkpoints_t), intent(inout) :: eio
     type(event_t), intent(in), target :: event
     integer, intent(in) :: i_prc
     logical, intent(in), optional :: reading
     logical :: rd
     rd = .false.;  if (present (reading))  rd = reading
     if (eio%active) then
        if (.not. eio%running)  call eio%startup ()
        if (eio%running) then
           eio%i_evt = eio%i_evt + 1
           if (rd) then
              eio%n_read = eio%n_read + 1
           else if (mod (eio%i_evt, eio%val) == 0) then
              call eio%message (eio%blank)
           end if
           if (eio%i_evt == eio%n_events)  call eio%shutdown ()
        end if
     end if
   end subroutine eio_checkpoints_output
```

When the first event is called, we have to initialize the screen output.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
   procedure :: startup => eio_checkpoints_startup
```

⟨*EIO checkpoints: procedures*⟩+≡
```
   subroutine eio_checkpoints_startup (eio)
     class(eio_checkpoints_t), intent(inout) :: eio
     if (eio%active .and. eio%i_evt < eio%n_events) then
        call msg_message ("")
        call msg_message (checkpoint_bar)
        call msg_message (checkpoint_head)
        call msg_message (checkpoint_bar)
        write (msg_buffer, checkpoint_fmt) 0., 0, eio%n_events - eio%i_evt, "???"
        call msg_message ()
        eio%running = .true.
        call eio%timer%start ()
     end if
   end subroutine eio_checkpoints_startup
```

This message is printed at every checkpoint.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
   procedure :: message => eio_checkpoints_message
```

⟨*EIO checkpoints: procedures*⟩+≡

```
    subroutine eio_checkpoints_message (eio, testflag)
      class(eio_checkpoints_t), intent(inout) :: eio
      logical, intent(in), optional :: testflag
      real :: t
      type(time_t) :: time_remaining
      type(string_t) :: time_string
      call eio%timer%stop ()
      t = eio%timer
      call eio%timer%restart ()
      time_remaining = &
           nint (t / (eio%i_evt - eio%n_read) * (eio%n_events - eio%i_evt))
      time_string = time_remaining%to_string_ms (blank = testflag)
      write (msg_buffer, checkpoint_fmt) &
           100 * (eio%i_evt - eio%n_read) / real (eio%n_events - eio%n_read), &
           eio%i_evt - eio%n_read, &
           eio%n_events - eio%i_evt, &
           char (time_string)
      call msg_message ()
    end subroutine eio_checkpoints_message
```

When the last event is called, wrap up.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
    procedure :: shutdown => eio_checkpoints_shutdown
```

⟨*EIO checkpoints: procedures*⟩+≡

```
    subroutine eio_checkpoints_shutdown (eio)
      class(eio_checkpoints_t), intent(inout) :: eio
      if (mod (eio%i_evt, eio%val) /= 0) then
         write (msg_buffer, checkpoint_fmt) &
              100., eio%i_evt - eio%n_read, 0, "0m:00s"
         call msg_message ()
      end if
      call msg_message (checkpoint_bar)
      call msg_message ("")
      eio%running = .false.
    end subroutine eio_checkpoints_shutdown
```

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
    procedure :: input_i_prc => eio_checkpoints_input_i_prc
    procedure :: input_event => eio_checkpoints_input_event
```

⟨*EIO checkpoints: procedures*⟩+≡

```
    subroutine eio_checkpoints_input_i_prc (eio, i_prc, iostat)
      class(eio_checkpoints_t), intent(inout) :: eio
      integer, intent(out) :: i_prc
      integer, intent(out) :: iostat
      call msg_bug ("Event checkpoints: event input not supported")
      i_prc = 0
      iostat = 1
    end subroutine eio_checkpoints_input_i_prc

    subroutine eio_checkpoints_input_event (eio, event, iostat)
      class(eio_checkpoints_t), intent(inout) :: eio
```

```
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      call msg_bug ("Event checkpoints: event input not supported")
      iostat = 1
    end subroutine eio_checkpoints_input_event
```

19.4.4 Message header

⟨*EIO checkpoints: parameters*⟩≡
```
    character(*), parameter :: &
       checkpoint_head = "| % complete | events generated | events remaining &
       &| time remaining"
    character(*), parameter :: &
       checkpoint_bar  = "|=================================================&
       &=================|"
    character(*), parameter :: &
       checkpoint_fmt  = "('    ',F5.1,T16,I9,T35,I9,T58,A)"
```

19.4.5 Unit tests

⟨*EIO checkpoints: public*⟩+≡
```
    public :: eio_checkpoints_test
```

⟨*EIO checkpoints: tests*⟩≡
```
    subroutine eio_checkpoints_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*EIO checkpoints: execute tests*⟩
```
    end subroutine eio_checkpoints_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO checkpoints: execute tests*⟩≡
```
    call test (eio_checkpoints_1, "eio_checkpoints_1", &
         "read and write event contents", &
         u, results)
```
⟨*EIO checkpoints: tests*⟩+≡
```
    subroutine eio_checkpoints_1 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      class(eio_t), allocatable :: eio
      type(event_sample_data_t) :: data
      type(string_t) :: sample
      integer :: i, n_events
```

```
write (u, "(A)")  "* Test output: eio_checkpoints_1"
write (u, "(A)")  "*   Purpose: generate a number of events &
     &with screen output"
write (u, "(A)")

call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

write (u, "(A)")
write (u, "(A)")  "* Generate events"
write (u, "(A)")

sample = "eio_checkpoints_1"

allocate (eio_checkpoints_t :: eio)

n_events = 10
call data%init (1, 0)
data%n_evt = n_events

select type (eio)
type is (eio_checkpoints_t)
   call eio%set_parameters (checkpoint = 4)
end select

call eio%init_out (sample, [process_ptr], data)

do i = 1, n_events
   call event%generate (1, [0._default, 0._default])
   call eio%output (event, i_prc = 0)
end do

write (u, "(A)")  "* Checkpointing status"
write (u, "(A)")

call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)
```

```
      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_checkpoints_1"

    end subroutine eio_checkpoints_1
```

19.5 HEP Common Blocks

Long ago, to transfer data between programs one had to set up a common
block and link both programs as libraries to the main executable. The HEP
community standardizes several of those common blocks.

The modern way of data exchange uses data files with standard formats.
However, the LHEF standard data format derives from a common block (actu-
ally, two).

WHIZARD used to support those common blocks, and LHEF was implemented
via writing/reading blocks. We still keep this convention, but intend to eliminate
common blocks (or any other static storage) from the workflow in the future.
This will gain flexibility towards concurrent running of program images.

We encapsulate everything here in a module. The module holds the variables
which are part of the common block.

Note: This code is taken essentially unchanged from WHIZARD 2.1 and does
not (yet) provide unit tests.

⟨hep_common.f90⟩≡
  ⟨*File header*⟩

  module hep_common

    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use diagnostics !NODEP!

    use lorentz !NODEP!
    use flavors
    use colors
    use polarizations
    use models
    use particles
    use subevents
    use processes

  ⟨*Standard module head*⟩

⟨*HEP common: public*⟩

⟨*HEP common: interfaces*⟩

⟨*HEP common: parameters*⟩

⟨*HEP common: variables*⟩

⟨*HEP common: common blocks*⟩

```
contains
```

⟨*HEP common: procedures*⟩

```
end module hep_common
```

19.5.1 Event characteristics

The maximal number of particles in an event record.

⟨*HEP common: parameters*⟩≡
```
    integer, parameter :: MAXNUP = 500
```

The number of particles in this event.

⟨*HEP common: variables*⟩≡
```
    integer :: NUP
```

The process ID for this event.

⟨*HEP common: variables*⟩+≡
```
    integer :: IDPRUP
```

The weight of this event (± 1 for unweighted events).

⟨*HEP common: variables*⟩+≡
```
    double precision :: XWGTUP
```

The factorization scale that is used for PDF calculation (-1 if undefined).

⟨*HEP common: variables*⟩+≡
```
    double precision :: SCALUP
```

The QED and QCD couplings α used for this event (-1 if undefined).

⟨*HEP common: variables*⟩+≡
```
    double precision :: AQEDUP
    double precision :: AQCDUP
```

19.5.2 Particle characteristics

The PDG code:

⟨*HEP common: variables*⟩+≡
```
    integer, dimension(MAXNUP) :: IDUP
```

The status code. Incoming: -1, outgoing: $+1$. Intermediate t-channel propagator: -2 (currently not used by WHIZARD). Intermediate resonance whose mass should be preserved: 2. Intermediate resonance for documentation: 3 (currently not used). Beam particles: -9.

⟨*HEP common: variables*⟩+≡
```
    integer, dimension(MAXNUP) :: ISTUP
```

Index of first and last mother.

⟨*HEP common: variables*⟩+≡
```
    integer, dimension(2,MAXNUP) :: MOTHUP
```

Color line index of the color and anticolor entry for the particle. The standard recommends using large numbers; we start from MAXNUP+1.

⟨*HEP common: variables*⟩+≡
```
    integer, dimension(2,MAXNUP) :: ICOLUP
```

Momentum, energy, and invariant mass: (p_x, p_y, p_z, E, M). For space-like particles, M is the negative square root of the absolute value of the invariant mass.

⟨*HEP common: variables*⟩+≡
```
    double precision, dimension(5,MAXNUP) :: PUP
```

Invariant lifetime (distance) from production to decay in mm.

⟨*HEP common: variables*⟩+≡
```
    double precision, dimension(MAXNUP) :: VTIMUP
```

Cosine of the angle between the spin-vector and a particle and the 3-momentum of its mother, given in the lab frame. If undefined/unpolarized: 9.

⟨*HEP common: variables*⟩+≡
```
    double precision, dimension(MAXNUP) :: SPINUP
```

19.5.3   The HEPRUP common block

This common block is filled once per run.

Run characteristics

The maximal number of different processes.

⟨*HEP common: parameters*⟩+≡
```
    integer, parameter :: MAXPUP = 100
```

The beam PDG codes.

⟨*HEP common: variables*⟩+≡
```
    integer, dimension(2) :: IDBMUP
```

The beam energies in GeV.

⟨*HEP common: variables*⟩+≡
```
    double precision, dimension(2) :: EBMUP
```

The PDF group and set for the two beams. (Undefined: use −1; LHAPDF: use group = 0).

⟨*HEP common: variables*⟩+≡
```
    integer, dimension(2) :: PDFGUP
    integer, dimension(2) :: PDFSUP
```

The (re)weighting model. 1: events are weighted, the shower generator (SHG) selects processes according to the maximum weight (in pb) and unweights events. 2: events are weighted, the SHG selects processes according to their cross section (in pb) and unweights events. 3: events are unweighted and simply run through the SHG. 4: events are weighted, and the SHG keeps the weight. Negative numbers: negative weights are allowed (and are reweighted to ±1 by the SHG, if allowed).

WHIZARD only supports modes 3 and 4, as the SHG is not given control over process selection. This is consistent with writing events to file, for offline showering.

⟨*HEP common: variables*⟩+≡
```
integer :: IDWTUP
```

The number of different processes.

⟨*HEP common: variables*⟩+≡
```
integer :: NPRUP
```

Process characteristics

Cross section and error in pb. (Cross section is needed only for `IDWTUP` = 2, so here both values are given for informational purposes only.)

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(MAXPUP) :: XSECUP
double precision, dimension(MAXPUP) :: XERRUP
```

Maximum weight, i.e., the maximum value that `XWGTUP` can take. Also unused for the supported weighting models. It is ±1 for unweighted events.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(MAXPUP) :: XMAXUP
```

Internal ID of the selected process, matches `IDPRUP` below.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(MAXPUP) :: LPRUP
```

The common block

⟨*HEP common: common blocks*⟩≡
```
common /HEPRUP/ &
     IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP, &
     XSECUP, XERRUP, XMAXUP, LPRUP
save /HEPRUP/
```

Fill the run characteristics of the common block. The initialization sets the beam properties, number of processes, and weighting model.

⟨*HEP common: public*⟩≡
```
public :: heprup_init
```

⟨*HEP common: procedures*⟩≡
```
subroutine heprup_init &
     (beam_pdg, beam_energy, n_processes, unweighted, negative_weights)
  integer, dimension(2), intent(in) :: beam_pdg
  real(default), dimension(2), intent(in) :: beam_energy
  integer, intent(in) :: n_processes
  logical, intent(in) :: unweighted
  logical, intent(in) :: negative_weights
  IDBMUP = beam_pdg
  EBMUP = beam_energy
  PDFGUP = -1
  PDFSUP = -1
  if (unweighted) then
     IDWTUP = 3
```

```
      else
         IDWTUP = 4
      end if
      if (negative_weights)  IDWTUP = - IDWTUP
      NPRUP = n_processes
   end subroutine heprup_init
```

Extract parameters from the common block. We leave it to the caller to specify which parameters it actually needs.

PDFGUP and PDFSUP are not extracted. IDWTUP=1,2 are not supported by WHIZARD, but correspond to weighted events.

⟨*HEP common: public*⟩+≡
```
   public :: heprup_get_run_parameters
```

⟨*HEP common: procedures*⟩+≡
```
   subroutine heprup_get_run_parameters &
        (beam_pdg, beam_energy, n_processes, unweighted, negative_weights)
     integer, dimension(2), intent(out), optional :: beam_pdg
     real(default), dimension(2), intent(out), optional :: beam_energy
     integer, intent(out), optional :: n_processes
     logical, intent(out), optional :: unweighted
     logical, intent(out), optional :: negative_weights
     if (present (beam_pdg))  beam_pdg = IDBMUP
     if (present (beam_energy))  beam_energy = EBMUP
     if (present (n_processes))  n_processes = NPRUP
     if (present (unweighted)) then
        select case (abs (IDWTUP))
        case (3)
           unweighted = .true.
        case (4)
           unweighted = .false.
        case (1,2)  ! not supported by WHIZARD
           unweighted = .false.
        case default
           call msg_fatal ("HEPRUP: unsupported IDWTUP value")
        end select
     end if
     if (present (negative_weights)) then
        negative_weights = IDWTUP < 0
     end if
   end subroutine heprup_get_run_parameters
```

Specify PDF set info. Since we support only LHAPDF, the group entry is zero.

⟨*HEP common: public*⟩+≡
```
   public :: heprup_set_lhapdf_id
```

⟨*HEP common: procedures*⟩+≡
```
   subroutine heprup_set_lhapdf_id (i_beam, pdf_id)
     integer, intent(in) :: i_beam, pdf_id
     PDFGUP(i_beam) = 0
     PDFSUP(i_beam) = pdf_id
   end subroutine heprup_set_lhapdf_id
```

Fill the characteristics for a particular process. Only the process ID is mandatory. Note that WHIZARD computes cross sections in fb, so we have to rescale to pb. The maximum weight is meaningless for unweighted events.

⟨*HEP common: public*⟩+≡
```
    public :: heprup_set_process_parameters
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine heprup_set_process_parameters &
         (i, process_id, cross_section, error, max_weight)
      integer, intent(in) :: i, process_id
      real(default), intent(in), optional :: cross_section, error, max_weight
      real(default), parameter :: pb_per_fb = 1.e-3_default
      LPRUP(i) = process_id
      if (present (cross_section)) then
         XSECUP(i) = cross_section * pb_per_fb
      else
         XSECUP(i) = 0
      end if
      if (present (error)) then
         XERRUP(i) = error * pb_per_fb
      else
         XERRUP(i) = 0
      end if
      select case (IDWTUP)
      case (3);  XMAXUP(i) = 1
      case (4)
         if (present (max_weight)) then
            XMAXUP(i) = max_weight * pb_per_fb
         else
            XMAXUP(i) = 0
         end if
      end select
    end subroutine heprup_set_process_parameters
```

Extract the process parameters, as far as needed.

⟨*HEP common: public*⟩+≡
```
    public :: heprup_get_process_parameters
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine heprup_get_process_parameters  &
         (i, process_id, cross_section, error, max_weight)
      integer, intent(in) :: i
      integer, intent(out), optional :: process_id
      real(default), intent(out), optional :: cross_section, error, max_weight
      real(default), parameter :: pb_per_fb = 1.e-3_default
      if (present (process_id))  process_id = LPRUP(i)
      if (present (cross_section)) then
         cross_section = XSECUP(i) / pb_per_fb
      end if
      if (present (error)) then
         error = XERRUP(i) / pb_per_fb
      end if
      if (present (max_weight)) then
         select case (IDWTUP)
```

```
       case (3)
          max_weight = 1
       case (4)
          max_weight = XMAXUP(i) / pb_per_fb
       case (1,2)   ! not supported by WHIZARD
          max_weight = 0
       case default
          call msg_fatal ("HEPRUP: unsupported IDWTUP value")
       end select
    end if
end subroutine heprup_get_process_parameters
```

19.5.4   Run parameter output (verbose)

This is a verbose output of the HEPRUP block.

⟨*HEP common: public*⟩+≡
```
    public :: heprup_write_verbose
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine heprup_write_verbose (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "HEPRUP Common Block"
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "IDBMUP", IDBMUP, &
           "PDG code of beams"
      write (u, "(3x,A6,' = ',G12.5,1x,G12.5,8x,A)")  "EBMUP ", EBMUP, &
           "Energy of beams in GeV"
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "PDFGUP", PDFGUP, &
           "PDF author group [-1 = undefined]"
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "PDFSUP", PDFSUP, &
           "PDF set ID      [-1 = undefined]"
      write (u, "(3x,A6,' = ',I9,3x,1x,9x,3x,8x,A)")  "IDWTUP", IDWTUP, &
           "LHA code for event weight mode"
      write (u, "(3x,A6,' = ',I9,3x,1x,9x,3x,8x,A)")  "NPRUP ", NPRUP, &
           "Number of user subprocesses"
      do i = 1, NPRUP
         write (u, "(1x,A,I0)")  "Subprocess #", i
         write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "XSECUP", XSECUP(i), &
              "Cross section in pb"
         write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "XERRUP", XERRUP(i), &
              "Cross section error in pb"
         write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "XMAXUP", XMAXUP(i), &
              "Maximum event weight (cf. IDWTUP)"
         write (u, "(3x,A6,' = ',I9,3x,1x,12x,8x,A)")  "LPRUP ", LPRUP(i), &
              "Subprocess ID"
      end do
    end subroutine heprup_write_verbose
```

19.5.5 Run parameter output (other formats)

This routine writes the initialization block according to the LHEF standard. It uses the current contents of the HEPRUP block.

⟨*HEP common: public*⟩+≡
```
    public :: heprup_write_lhef
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine heprup_write_lhef (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(2(1x,I0),2(1x,ES17.10),6(1x,I0))") &
           IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
      do i = 1, NPRUP
         write (u, "(3(1x,ES17.10),1x,I0)") &
              XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
      end do
    end subroutine heprup_write_lhef
```

This routine is a complete dummy at the moment. It uses the current contents of the HEPRUP block. At the end, it should depend on certain input flags for the different ASCII event formats.

⟨*HEP common: public*⟩+≡
```
    public :: heprup_write_ascii
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine heprup_write_ascii (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(2(1x,I0),2(1x,ES17.10),6(1x,I0))") &
           IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
      do i = 1, NPRUP
         write (u, "(3(1x,ES17.10),1x,I0)") &
              XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
      end do
    end subroutine heprup_write_ascii
```

Run parameter input (LHEF)

In a LHEF file, the parameters are written in correct order on separate lines, but we should not count on the precise format. List-directed input should just work.

⟨*HEP common: public*⟩+≡
```
    public :: heprup_read_lhef
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine heprup_read_lhef (u)
      integer, intent(in) :: u
      integer :: i
      read (u, *) &
           IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
```

2565

```
      do i = 1, NPRUP
         read (u, *) &
              XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
      end do
   end subroutine heprup_read_lhef
```

19.5.6   The HEPEUP common block

⟨*HEP common: common blocks*⟩+≡
```
   common /HEPEUP/ &
        NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP, &
        IDUP, ISTUP, MOTHUP, ICOLUP, PUP, VTIMUP, SPINUP
   save /HEPEUP/
```

Initialization

Fill the event characteristics of the common block. The initialization sets only
the number of particles and initializes the rest with default values. The other
routine sets the optional parameters.

⟨*HEP common: public*⟩+≡
```
   public :: hepeup_init
   public :: hepeup_set_event_parameters
```

⟨*HEP common: procedures*⟩+≡
```
   subroutine hepeup_init (n_tot)
     integer, intent(in) :: n_tot
     NUP = n_tot
     IDPRUP = 0
     XWGTUP = 1
     SCALUP = -1
     AQEDUP = -1
     AQCDUP = -1
   end subroutine hepeup_init

   subroutine hepeup_set_event_parameters &
       (proc_id, weight, scale, alpha_qed, alpha_qcd)
     integer, intent(in), optional :: proc_id
     real(default), intent(in), optional :: weight, scale, alpha_qed, alpha_qcd
     if (present (proc_id))   IDPRUP = proc_id
     if (present (weight))    XWGTUP = weight
     if (present (scale))     SCALUP = scale
     if (present (alpha_qed)) AQEDUP = alpha_qed
     if (present (alpha_qcd)) AQCDUP = alpha_qcd
   end subroutine hepeup_set_event_parameters
```

Extract event information. The caller determines the parameters.

⟨*HEP common: public*⟩+≡
```
   public :: hepeup_get_event_parameters
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepeup_get_event_parameters &
         (proc_id, weight, scale, alpha_qed, alpha_qcd)
      integer, intent(out), optional :: proc_id
      real(default), intent(out), optional :: weight, scale, alpha_qed, alpha_qcd
      if (present (proc_id))   proc_id   = IDPRUP
      if (present (weight))    weight    = XWGTUP
      if (present (scale))     scale     = SCALUP
      if (present (alpha_qed)) alpha_qed = AQEDUP
      if (present (alpha_qcd)) alpha_qcd = AQCDUP
    end subroutine hepeup_get_event_parameters
```

Particle data

Below we need the particle status codes which are actually defined in the `subevents` module.

   Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

⟨*HEP common: public*⟩+≡
```
    public :: hepeup_set_particle
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepeup_set_particle (i, pdg, status, parent, col, p, m2)
      integer, intent(in) :: i
      integer, intent(in) :: pdg, status
      integer, dimension(:), intent(in) :: parent
      type(vector4_t), intent(in) :: p
      integer, dimension(2), intent(in) :: col
      real(default), intent(in) :: m2
      if (i > MAXNUP) then
         call msg_error (arr=[ &
              var_str ("Too many particles in HEPEUP common block. " // &
                          "If this happened "), &
              var_str ("during event output, your events will be " // &
                          "invalid; please consider "), &
              var_str ("switching to a modern event format like HEPMC. " // &
                          "If you are not "), &
              var_str ("using an old, HEPEUP based format and " // &
                          "nevertheless get this error,"), &
              var_str ("please notify the WHIZARD developers,") ])
         return
      end if
      IDUP(i) = pdg
      select case (status)
      case (PRT_BEAM);         ISTUP(i) = -9
      case (PRT_INCOMING);     ISTUP(i) = -1
      case (PRT_BEAM_REMNANT); ISTUP(i) =  3
      case (PRT_OUTGOING);     ISTUP(i) =  1
      case (PRT_RESONANT);     ISTUP(i) =  2
      case (PRT_VIRTUAL);      ISTUP(i) =  3
      case default;            ISTUP(i) =  0
      end select
      select case (size (parent))
```

```
      case (0);      MOTHUP(:,i) = 0
      case (1);      MOTHUP(1,i) = parent(1); MOTHUP(2,i) = 0
      case default;  MOTHUP(:,i) = [ parent(1), parent(size (parent)) ]
      end select
      if (col(1) > 0) then
         ICOLUP(1,i) = 500 + col(1)
      else
         ICOLUP(1,i) = 0
      end if
      if (col(2) > 0) then
         ICOLUP(2,i) = 500 + col(2)
      else
         ICOLUP(2,i) = 0
      end if
      PUP(1:3,i) = vector3_get_components (space_part (p))
      PUP(4,i) = energy (p)
      PUP(5,i) = sign (sqrt (abs (m2)), m2)
      VTIMUP(i) = 0
      SPINUP(i) = 9
   end subroutine hepeup_set_particle
```

Set the lifetime, actually $c\tau$ measured im mm, where τ is the invariant lifetime.

⟨*HEP common: public*⟩+≡
```
   public :: hepeup_set_particle_lifetime
```

⟨*HEP common: procedures*⟩+≡
```
   subroutine hepeup_set_particle_lifetime (i, lifetime)
     integer, intent(in) :: i
     real(default), intent(in) :: lifetime
     VTIMUP(i) = lifetime
   end subroutine hepeup_set_particle_lifetime
```

Set the particle spin entry. We need the cosine of the angle of the spin axis with respect to the three-momentum of the parent particle.

If the particle has a full polarization density matrix given, we need the particle momentum and polarization as well as the mother-particle momentum. The polarization is transformed into a spin vector (which is sensible only for spin-1/2 or massless particles), which then is transformed into the lab frame (by a rotation of the 3-axis to the particle momentum axis). Finally, we compute the scalar product of this vector with the mother-particle three-momentum.

This puts severe restrictions on the applicability of this definition, and Lorentz invariance is lost. Unfortunately, the Les Houches Accord requires this computation.

⟨*HEP common: public*⟩+≡
```
   public :: hepeup_set_particle_spin
```

⟨*HEP common: interfaces*⟩≡
```
   interface hepeup_set_particle_spin
      module procedure hepeup_set_particle_spin_pol
   end interface
```

⟨*HEP common: procedures*⟩+≡
```
   subroutine hepeup_set_particle_spin_pol (i, p, pol, p_mother)
```

```
      integer, intent(in) :: i
      type(vector4_t), intent(in) :: p
      type(polarization_t), intent(in) :: pol
      type(vector4_t), intent(in) :: p_mother
      type(vector3_t) :: s3, p3
      type(vector4_t) :: s4
      s3 = vector3_moving (polarization_get_axis (pol))
      p3 = space_part (p)
      s4 = rotation_to_2nd (3, p3) * vector4_moving (0._default, s3)
      SPINUP(i) = enclosed_angle_ct (s4, p_mother)
   end subroutine hepeup_set_particle_spin_pol
```

Extract particle data. The caller decides which ones to retrieve.

Status codes: beam remnants share the status code with virtual particles. However, for the purpose of WHIZARD we should identify them. We use the PDG code for this.

⟨*HEP common: public*⟩+≡
```
    public :: hepeup_get_particle
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepeup_get_particle (i, pdg, status, parent, col, p, m2)
      integer, intent(in) :: i
      integer, intent(out), optional :: pdg, status
      integer, dimension(:), intent(out), optional :: parent
      type(vector4_t), intent(out), optional :: p
      integer, dimension(2), intent(out), optional :: col
      real(default), dimension(5,MAXNUP) :: pup_def
      real(default), intent(out), optional :: m2
      if (present (pdg))  pdg = IDUP(i)
      if (present (status)) then
         select case (ISTUP(i))
         case (-9);  status = PRT_BEAM
         case (-1);  status = PRT_INCOMING
         case (1);   status = PRT_OUTGOING
         case (2);   status = PRT_RESONANT
         case (3);
            select case (IDUP(i))
            case (HADRON_REMNANT, HADRON_REMNANT_SINGLET, &
                  HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET)
               status = PRT_BEAM_REMNANT
            case default
               status = PRT_VIRTUAL
            end select
         case default
            status = PRT_UNDEFINED
         end select
      end if
      if (present (parent)) then
         select case (size (parent))
         case (0)
         case (1);   parent(1) = MOTHUP(1,i)
         case (2);   parent = MOTHUP(:,i)
         end select
      end if
```

```
   if (present (col)) then
      col = ICOLUP(:,i)
   end if
   if (present (p)) then
      pup_def = PUP
      p = vector4_moving (pup_def(4,i), vector3_moving (pup_def(1:3,i)))
   end if
   if (present (m2)) then
      m2 = sign (PUP(5,i) ** 2, PUP(5,i))
   end if
end subroutine hepeup_get_particle
```

19.5.7 The HEPEVT common block

For the LEP Monte Carlos, a standard common block has been proposed in
AKV89. We strongly recommend its use. (The description is an abbreviated
transcription of AKV89, Vol. 3, pp. 327-330).

NMXHEP is the maximum number of entries:

⟨*HEP common: variables*⟩+≡
```
   integer, parameter :: NMXHEP = 4000
```

NEVHEP is normally the event number, but may take special values as follows:

0 the program does not keep track of event numbers. -1 a special initializa-
tion record. -2 a special final record.

⟨*HEP common: variables*⟩+≡
```
   integer :: NEVHEP
```

NHEP holds the number of entries for this event.

⟨*HEP common: variables*⟩+≡
```
   integer :: NHEP
```

The entry ISTHEP(N) gives the status code for the Nth entry, with the following
semantics: 0 a null entry. 1 an existing entry, which has not decayed or frag-
mented. 2 a decayed or fragmented entry, which is retained for event history
information. 3 documentation line. 4- 10 reserved for future standards. 11-200
at the disposal of each model builder. 201- at the disposal of users.

⟨*HEP common: variables*⟩+≡
```
   integer, dimension(NMXHEP) :: ISTHEP
```

The Particle Data Group has proposed standard particle codes, which are to be
stored in IDHEP(N).

⟨*HEP common: variables*⟩+≡
```
   integer, dimension(NMXHEP) :: IDHEP
```

JMOHEP(1,N) points to the mother of the Nth entry, if any. It is set to zero for
initial entries. JMOHEP(2,N) points to the second mother, if any.

⟨*HEP common: variables*⟩+≡
```
   integer, dimension(2, NMXHEP) :: JMOHEP
```

`JDAHEP(1,N)` and `JDAHEP(2,N)` point to the first and last daughter of the `N`th entry, if any. These are zero for entries which have not yet decayed. The other daughters are stored in between these two.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2, NMXHEP) :: JDAHEP
```

In `PHEP` we store the momentum of the particle, more specifically this means that `PHEP(1,N)`, `PHEP(2,N)`, and `PHEP(3,N)` contain the momentum in the x, y, and z direction (as defined by the machine people), measured in GeV/c. `PHEP(4,N)` contains the energy in GeV and `PHEP(5,N)` the mass in GeV/c^2. The latter may be negative for spacelike partons.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(5, NMXHEP) :: PHEP
```

Finally `VHEP` is the place to store the position of the production vertex. `VHEP(1,N)`, `VHEP(2,N)`, and `VHEP(3,N)` contain the x, y, and z coordinate (as defined by the machine people), measured in mm. `VHEP(4,N)` contains the production time in mm/c.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(4, NMXHEP) :: VHEP
```

As an amendment to the proposed standard common block HEPEVT, we also have a polarisation common block HEPSPN, as described in AKV89. `SHEP(1,N)`, `SHEP(2,N)`, and `SHEP(3,N)` give the x, y, and z component of the spinvector s of a fermion in the fermions restframe.

Furthermore, we add the polarization of the corresponding outgoing particles:

⟨*HEP common: variables*⟩+≡
```
integer, dimension(NMXHEP) :: hepevt_pol
```

By convention, `SHEP(4,N)` is always 1. All this is taken from StdHep 4.06 manual and written using Fortran90 conventions.

⟨*HEP common: common blocks*⟩+≡
```
common /HEPEVT/ &
     NEVHEP, NHEP, ISTHEP, IDHEP, &
     JMOHEP, JDAHEP, PHEP, VHEP
save /HEPEVT/
```

Here we store HEPEVT parameters of the WHIZARD 1 realization which are not part of the HEPEVT common block.

⟨*HEP common: variables*⟩+≡
```
integer :: hepevt_n_out, hepevt_n_remnants
```

⟨*HEP common: variables*⟩+≡
```
double precision :: hepevt_weight, hepevt_function_value
double precision :: hepevt_function_ratio
```

Filling HEPEVT: If the event count is not provided, set `NEVHEP` to zero. If the event count is `-1` or `-2`, the record corresponds to initialization and finalization, and the event is irrelevant.

Note that the event count may be larger than 2^{31} (2 GEvents). In that case, cut off the upper bits since `NEVHEP` is probably limited to default integer.

⟨*HEP common: public*⟩+≡
```
    public :: hepevt_init
    public :: hepevt_set_event_parameters
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepevt_init (n_tot, n_out)
      integer, intent(in) :: n_tot, n_out
      NHEP              = n_tot
      NEVHEP            = 0
      hepevt_n_out      = n_out
      hepevt_n_remnants = 0
      hepevt_weight     = 1
      hepevt_function_value = 0
      hepevt_function_ratio = 1
    end subroutine hepevt_init

    subroutine hepevt_set_event_parameters &
        (weight, function_value, function_ratio, i_evt)
      integer, intent(in), optional :: i_evt
      real(default), intent(in), optional :: weight, function_value, &
        function_ratio
      if (present (i_evt)) NEVHEP = i_evt
      if (present (weight)) hepevt_weight = weight
      if (present (function_value)) hepevt_function_value = &
          function_value
      if (present (function_ratio)) hepevt_function_ratio = &
          function_ratio
    end subroutine hepevt_set_event_parameters
```

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

⟨*HEP common: public*⟩+≡
```
    public :: hepevt_set_particle
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepevt_set_particle (i, pdg, status, parent, child, p, m2, hel)
      integer, intent(in) :: i
      integer, intent(in) :: pdg, status
      integer, dimension(:), intent(in) :: parent
      integer, dimension(:), intent(in) :: child
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: m2
      integer, intent(in) :: hel
      IDHEP(i) = pdg
      select case (status)
        case (PRT_BEAM);     ISTHEP(i) = 2
        case (PRT_INCOMING); ISTHEP(i) = 2
        case (PRT_OUTGOING); ISTHEP(i) = 1
        case (PRT_VIRTUAL);  ISTHEP(i) = 2
```

```
      case (PRT_RESONANT);  ISTHEP(i) = 2
      case default;         ISTHEP(i) = 0
    end select
    select case (size (parent))
    case (0);        JMOHEP(:,i) = 0
    case (1);        JMOHEP(1,i) = parent(1); JMOHEP(2,i) = 0
    case default;  JMOHEP(:,i) = [ parent(1), parent(size (parent)) ]
    end select
    select case (size (child))
    case (0);        JDAHEP(:,i) = 0
    case (1);        JDAHEP(:,i) = child(1)
    case default;  JDAHEP(:,i) = [ child(1), child(size (child)) ]
    end select
    PHEP(1:3,i) = vector3_get_components (space_part (p))
    PHEP(4,i) = energy (p)
    PHEP(5,i) = sign (sqrt (abs (m2)), m2)
    VHEP(1:4,i) = 0
    hepevt_pol(i) = hel
  end subroutine hepevt_set_particle
```

19.5.8   Event output

This is a verbose output of the HEPEVT block.

⟨*HEP common: public*⟩+≡
```
    public :: hepevt_write_verbose
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepevt_write_verbose (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "HEPEVT Common Block"
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "NEVHEP", NEVHEP, &
           "Event number"
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "NHEP  ", NHEP, &
           "Number of particles in event"
      do i = 1, NHEP
         write (u, "(1x,A,I0)")  "Particle #", i
         write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)", advance="no") &
              "ISTHEP", ISTHEP(i), "Status code: "
         select case (ISTHEP(i))
         case ( 0);  write (u, "(A)")  "null entry"
         case ( 1);  write (u, "(A)")  "outgoing"
         case ( 2);  write (u, "(A)")  "decayed"
         case ( 3);  write (u, "(A)")  "documentation"
         case (4:10);  write (u, "(A)")  "[unspecified]"
         case (11:200);  write (u, "(A)")  "[model-specific]"
         case (201:);  write (u, "(A)")  "[user-defined]"
         case default;  write (u, "(A)")  "[undefined]"
         end select
         write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "IDHEP ", IDHEP(i), &
              "PDG code of particle"
         write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "JMOHEP", JMOHEP(:,i), &
```

```
             "Index of first/second mother"
        write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "JDAHEP", JDAHEP(:,i), &
             "Index of first/last daughter"
        write (u, "(3x,A6,' = ',ES12.5,1x,ES12.5,8x,A)")  "PHEP12", &
             PHEP(1:2,i), "Transversal momentum (x/y) in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PHEP3 ", PHEP(3,i), &
             "Longitudinal momentum (z) in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PHEP4 ", PHEP(4,i), &
             "Energy in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PHEP5 ", PHEP(5,i), &
             "Invariant mass in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,ES12.5,8x,A)")  "VHEP12", VHEP(1:2,i), &
             "Transversal displacement (xy) in mm"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "VHEP3 ", VHEP(3,i), &
             "Longitudinal displacement (z) in mm"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "VHEP4 ", VHEP(4,i), &
             "Production time in mm"
      end do
    end subroutine hepevt_write_verbose
```

This is a verbose output of the HEPEUP block.

⟨*HEP common: public*⟩+≡
```
    public :: hepeup_write_verbose
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepeup_write_verbose (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(A)")  "HEPEUP Common Block"
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "NUP   ", NUP, &
           "Number of particles in event"
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "IDPRUP", IDPRUP, &
           "Subprocess ID"
      write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "XWGTUP", XWGTUP, &
           "Event weight"
      write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "SCALUP", SCALUP, &
           "Event energy scale in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "AQEDUP", AQEDUP, &
           "QED coupling [-1 = undefined]"
      write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "AQCDUP", AQCDUP, &
           "QCD coupling [-1 = undefined]"
      do i = 1, NUP
         write (u, "(1x,A,I0)")  "Particle #", i
         write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "IDUP  ", IDUP(i), &
              "PDG code of particle"
         write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)", advance="no") &
              "ISTUP ", ISTUP(i), "Status code: "
         select case (ISTUP(i))
         case (-1);  write (u, "(A)")  "incoming"
         case ( 1);  write (u, "(A)")  "outgoing"
         case (-2);  write (u, "(A)")  "spacelike"
         case ( 2);  write (u, "(A)")  "resonance"
         case ( 3);  write (u, "(A)")  "resonance (doc)"
```

```
        case (-9);  write (u, "(A)")  "beam"
        case default;  write (u, "(A)")  "[undefined]"
        end select
        write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "MOTHUP", MOTHUP(:,i), &
              "Index of first/last mother"
        write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "ICOLUP", ICOLUP(:,i), &
              "Color/anticolor flow index"
        write (u, "(3x,A6,' = ',ES12.5,1x,ES12.5,8x,A)")  "PUP1/2", PUP(1:2,i), &
              "Transversal momentum (x/y) in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PUP3  ", PUP(3,i), &
              "Longitudinal momentum (z) in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PUP4  ", PUP(4,i), &
              "Energy in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PUP5  ", PUP(5,i), &
              "Invariant mass in GeV"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "VTIMUP", VTIMUP(i), &
              "Invariant lifetime in mm"
        write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "SPINUP", SPINUP(i), &
              "cos(spin angle) [9 = undefined]"
      end do
    end subroutine hepeup_write_verbose
```

19.5.9 Event output in various formats

This routine writes event output according to the LHEF standard. It uses the current contents of the HEPEUP block.

⟨*HEP common: public*⟩+≡
```
    public :: hepeup_write_lhef
    public :: hepeup_write_lha
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepeup_write_lhef (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(2(1x,I0),4(1x,ES17.10))") &
            NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
      do i = 1, NUP
         write (u, "(6(1x,I0),7(1x,ES17.10))") &
               IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
               PUP(:,i), VTIMUP(i), SPINUP(i)
      end do
    end subroutine hepeup_write_lhef

    subroutine hepeup_write_lha (unit)
      integer, intent(in), optional :: unit
      integer :: u, i
      integer, dimension(MAXNUP) :: spin_up
      spin_up = SPINUP
      u = output_unit (unit);  if (u < 0)  return
      write (u, "(2(1x,I5),1x,ES17.10,3(1x,ES13.6))") &
            NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
      write (u, "(500(1x,I5))") IDUP(:NUP)
```

```
write (u, "(500(1x,I5))") MOTHUP(1,:NUP)
write (u, "(500(1x,I5))") MOTHUP(2,:NUP)
write (u, "(500(1x,I5))") ICOLUP(1,:NUP)
write (u, "(500(1x,I5))") ICOLUP(2,:NUP)
write (u, "(500(1x,I5))") ISTUP(:NUP)
write (u, "(500(1x,I5))") spin_up(:NUP)
do i = 1, NUP
        write (u, "(1x,I5,4(1x,ES17.10))") i, PUP([ 4,1,2,3 ], i)
end do

end subroutine hepeup_write_lha
```

This routine writes event output according to the HEPEVT standard. It uses
the current contents of the HEPEVT block and some additional parameters ac-
cording to the standard in WHIZARD 1. For the long ASCII format, the value
of the sample function (i.e. the product of squared matrix element, structure
functions and phase space factor is printed out). The option of reweighting ma-
trix elements with respect to some reference cross section is not implemented in
WHIZARD 2 for this event format, therefore the second entry in the long ASCII
format (the function ratio) is always one. The ATHENA format is an imple-
mentation of the HEPEVT format that is readable by the ATLAS ATHENA
software framework. It is very similar to the WHIZARD 1 HEPEVT format,
except that it contains an event counter, a particle counter inside the event, and
has the HEPEVT ISTHEP status before the PDG code. The MOKKA format
is a special ASCII format that contains the information to be parsed to the
MOKKA LC fast simulation software.

⟨*HEP common: public*⟩+≡
```
public :: hepevt_write_hepevt
public :: hepevt_write_ascii
public :: hepevt_write_athena
public :: hepevt_write_mokka
```

⟨*HEP common: procedures*⟩+≡
```
subroutine hepevt_write_hepevt (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(3(1x,I0),(1x,ES17.10))") &
       NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
     write (u, "(7(1x,I0))") &
          ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i), hepevt_pol(i)
     write (u, "(5(1x,ES17.10))") PHEP(:,i)
     write (u, "(5(1x,ES17.10))") VHEP(:,i), 0.d0
  end do
end subroutine hepevt_write_hepevt

subroutine hepevt_write_ascii (unit, long)
  integer, intent(in), optional :: unit
  logical, intent(in) :: long
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(3(1x,I0),(1x,ES17.10))") &
```

```
       NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
     write (u, "(2(1x,I0))") IDHEP(i), hepevt_pol(i)
     write (u, "(5(1x,ES17.10))") PHEP(:,i)
  end do
  if (long) then
     write (u, "(2(1x,ES17.10))") &
          hepevt_function_value, hepevt_function_ratio
  end if
end subroutine hepevt_write_ascii

subroutine hepevt_write_athena (unit, i_evt)
  integer, intent(in), optional :: unit, i_evt
  integer :: u, i, num_event
  num_event = 0
  if (present (i_evt)) num_event = i_evt
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(2(1x,I0))") num_event, NHEP
  do i = 1, NHEP
     write (u, "(7(1x,I0))") &
          i, ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i)
     write (u, "(5(1x,ES17.10))") PHEP(:,i)
     write (u, "(5(1x,ES17.10))") VHEP(1:4,i)
  end do
end subroutine hepevt_write_athena

subroutine hepevt_write_mokka (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  write (u, "(3(1x,I0),(1x,ES17.10))") &
       NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
     write (u, "(4(1x,I0),4(1x,ES17.10))") &
          ISTHEP(i), IDHEP(i), JDAHEP(1,i), JDAHEP(2,i), &
          PHEP(1:3,i), PHEP(5,i)
  end do
end subroutine hepevt_write_mokka
```

19.5.10 Event input in various formats

This routine writes event output according to the LHEF standard. It uses the current contents of the HEPEUP block.

⟨*HEP common: public*⟩+≡
```
    public :: hepeup_read_lhef
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepeup_read_lhef (u)
      integer, intent(in) :: u
      integer :: i
      read (u, *) &
           NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
      do i = 1, NUP
```

```
        read (u, *) &
             IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
             PUP(:,i), VTIMUP(i), SPINUP(i)
     end do
   end subroutine hepeup_read_lhef
```

19.5.11   Data Transfer: particle sets

The `WHIZARD` format for handling particle data in events is `particle_set_t`.
We have to interface this to the common blocks.

We first create a new particle set that contains only the particles that are
supported by the LHEF format. These are: beam, incoming, resonant, outgoing.
We drop particles with unknown, virtual or beam-remnant status.

From this set we fill the common block. Event information such as process
ID and weight is not transferred here; this has to be done by the caller. The spin
information is set only if the particle has a unique mother, and if its polarization
is fully defined.

⟨*HEP common: public*⟩+≡
```
   public :: hepeup_from_particle_set
```

⟨*HEP common: procedures*⟩+≡
```
   subroutine hepeup_from_particle_set (particle_set, keep_beams)
     type(particle_set_t), intent(in), target :: particle_set
     logical, intent(in), optional :: keep_beams
     type(particle_t), pointer :: prt
     type(particle_set_t), target :: pset_hepevt
     integer :: i, n_parents
     integer, dimension(1) :: i_mother
     call particle_set_to_hepevt_form (particle_set, pset_hepevt, keep_beams)
     call hepeup_init (pset_hepevt%n_tot)
     do i = 1, pset_hepevt%n_tot
        prt => pset_hepevt%prt(i)
        call hepeup_set_particle (i, &
             particle_get_pdg (prt), &
             particle_get_status (prt), &
             particle_get_parents (prt), &
             particle_get_color (prt), &
             particle_get_momentum (prt), &
             particle_get_p2 (prt))
        n_parents = particle_get_n_parents (prt)
        if (n_parents == 1) then
           i_mother = particle_get_parents (prt)
           select case (particle_get_polarization_status (prt))
           case (PRT_GENERIC_POLARIZATION)
              call hepeup_set_particle_spin (i, &
                   particle_get_momentum (prt), &
                   particle_get_polarization (prt), &
                   particle_get_momentum (pset_hepevt%prt(i_mother(1))))
           end select
        end if
     end do
     call particle_set_final (pset_hepevt)
```

```
      end subroutine hepeup_from_particle_set
```

Input. The particle set should be allocated properly, but we replace the particle contents.

If there are no beam particles in the event, we try to reconstruct beam particles and beam remnants. We assume for simplicity that the beam particles, if any, are the first two particles. If they are absent, the first two particles should be the incoming partons.

⟨*HEP common: public*⟩+≡
```
      public :: hepeup_to_particle_set
```

⟨*HEP common: procedures*⟩+≡
```
      subroutine hepeup_to_particle_set (particle_set, recover_beams, model)
        type(particle_set_t), intent(inout), target :: particle_set
        logical, intent(in), optional :: recover_beams
        type(model_t), intent(in), target :: model
        type(particle_t), dimension(:), allocatable :: prt
        integer, dimension(:), allocatable :: child
        integer :: i, j, k, pdg, status
        type(flavor_t) :: flv
        type(color_t) :: col
        integer, dimension(2) :: c
        type(vector4_t) :: p
        real(default) :: p2
        logical :: reconstruct
        integer :: off
        if (present (recover_beams)) then
           reconstruct = recover_beams .and. .not. all (ISTUP(1:2) == PRT_BEAM)
        else
           reconstruct = .false.
        end if
        if (reconstruct) then
           off = 4
        else
           off = 0
        end if
        allocate (prt (NUP + off), child (NUP + off))
        do i = 1, NUP
           k = i + off
           call hepeup_get_particle (i, pdg, status, col = c, p = p, m2 = p2)
           call flavor_init (flv, pdg, model)
           call particle_set_flavor (prt(k), flv)
           call particle_reset_status (prt(k), status)
           call color_init (col, c)
           call particle_set_color (prt(k), col)
           call particle_set_momentum (prt(k), p, p2)
           call particle_set_parents (prt(k), MOTHUP(:,i) + off)
           child = [(j, j = 1 + off, NUP + off)]
           where (MOTHUP(1,:NUP) /= i .and. MOTHUP(2,:NUP) /= i)  child = 0
           call particle_set_children (prt(k), child)
        end do
        if (reconstruct) then
           do k = 1, 2
              call particle_reset_status (prt(k), PRT_BEAM)
```

2579

```
                call particle_set_children (prt(k), [k+2,k+4])
            end do
            do k = 3, 4
                call particle_reset_status (prt(k), PRT_BEAM_REMNANT)
                call particle_set_parents (prt(k), [k-2])
            end do
            do k = 5, 6
                call particle_set_parents (prt(k), [k-4])
            end do
        else
            call handle_beams (prt)
        end if
        call particle_set_replace (particle_set, prt)
    end subroutine hepeup_to_particle_set
```

Beams and beam remnants require a special handling. `WHIZARD`'s interactions are set up such that in the list of children of beam particles, the remnant comes before the interacting parton. To enforce this, we may have to reorder things.

The beam particles will always be the first two particles.

⟨*HEP common: procedures*⟩+≡
```
    subroutine handle_beams (prt)
        type(particle_t), dimension(:), intent(inout) :: prt
        integer :: i
        integer, dimension(2) :: child
        do i = 1, 2
            select case (particle_get_status (prt(i)))
            case (PRT_BEAM)
                child = particle_get_children (prt(i))
                if (particle_get_status (prt(child(2))) == PRT_BEAM_REMNANT) then
                    call particle_set_children (prt(i), child([2,1]))
                end if
            end select
        end do
    end subroutine handle_beams
```

The HEPEVT common block is quite similar, but does contain less information, e.g. no color flows (it was LEP time). The spin information is set only if the particle has a unique mother, and if its polarization is fully defined.

⟨*HEP common: public*⟩+≡
```
    public :: hepevt_from_particle_set
```

⟨*HEP common: procedures*⟩+≡
```
    subroutine hepevt_from_particle_set (particle_set, keep_beams)
        type(particle_set_t), intent(in), target :: particle_set
        type(particle_t), pointer :: prt
        type(particle_set_t), target :: pset_hepevt
        logical, intent(in), optional :: keep_beams
        integer :: i
        call particle_set_to_hepevt_form (particle_set, pset_hepevt, keep_beams)
        call hepevt_init (pset_hepevt%n_tot, pset_hepevt%n_out)
        do i = 1, pset_hepevt%n_tot
            prt => pset_hepevt%prt(i)
            call hepevt_set_particle (i, &
```

```
            particle_get_pdg (prt), &
            particle_get_status (prt), &
            particle_get_parents (prt), &
            particle_get_children (prt), &
            particle_get_momentum (prt), &
            particle_get_p2 (prt), &
            particle_get_helicity (prt))
      end do
      call particle_set_final (pset_hepevt)
    end subroutine hepevt_from_particle_set
```

19.6   HEP Common and Events

This is a separate module that manages data exchange between the common
blocks and `event_t` objects. We separate this from the previous module in
order to avoid a circular module dependency.

⟨`hep_events.f90`⟩≡
  ⟨*File header*⟩

```
  module hep_events

    use kinds !NODEP!
```
⟨*Use file utils*⟩
```
     use diagnostics !NODEP!

    use models
    use particles
    use processes
    use hep_common
    use events
```

⟨*Standard module head*⟩

⟨*HEP events: public*⟩

```
  contains
```

⟨*HEP events: procedures*⟩

```
  end module hep_events
```

19.6.1   Data Transfer: events

Fill the HEPEUP block, given a `WHIZARD` event object.

⟨*HEP events: public*⟩≡
```
    public :: hepeup_from_event
```

⟨*HEP events: procedures*⟩≡
```
    subroutine hepeup_from_event (event, keep_beams, process_index)
      type(event_t), intent(in), target :: event
      logical, intent(in), optional :: keep_beams
```

```
        integer, intent(in), optional :: process_index
        type(particle_set_t), pointer :: particle_set
        real(default) :: scale, alpha_qcd
        if (event%has_particle_set ()) then
           particle_set => event%get_particle_set_ptr ()
           call hepeup_from_particle_set (particle_set, keep_beams)
           if (present (process_index)) &
                call hepeup_set_event_parameters (proc_id = process_index)
           scale = event%get_fac_scale ()
           if (scale /= 0) &
                call hepeup_set_event_parameters (scale = scale)
           alpha_qcd = event%get_alpha_s ()
           if (alpha_qcd /= 0) &
                call hepeup_set_event_parameters (alpha_qcd = alpha_qcd)
           if (event%weight_prc_is_known) then
              call hepeup_set_event_parameters (weight = event%weight_prc)
           else
              call msg_bug ("HEPEUP: process weight is unknown")
           end if
        else
           call msg_bug ("HEPEUP: event incomplete")
        end if
     end subroutine hepeup_from_event
```

Reverse.

Note: The current implementation sets the particle set of the hard process and is therefore not useful if the event on file is dressed. This should be reconsidered.

Note: setting of scale or alpha is not yet supported by the `event_t` object. Ticket #628.

⟨*HEP events: public*⟩+≡
```
     public :: hepeup_to_event
```

⟨*HEP events: procedures*⟩+≡
```
     subroutine hepeup_to_event (event, process_index, recover_beams)
       type(event_t), intent(inout), target :: event
       integer, intent(out), optional :: process_index
       logical, intent(in), optional :: recover_beams
       type(process_t), pointer :: process
       type(model_t), pointer :: model
       real(default) :: weight, scale, alpha_qcd
       type(particle_set_t) :: particle_set
       process => event%get_process_ptr ()
       model => process%get_model_ptr ()
       call hepeup_to_particle_set (particle_set, recover_beams, model)
       call event%set_particle_set_hard_proc (particle_set)
       call particle_set_final (particle_set)
       if (present (process_index)) then
          call hepeup_get_event_parameters (proc_id = process_index)
       end if
       call hepeup_get_event_parameters (weight = weight, &
            scale = scale, alpha_qcd = alpha_qcd)
       call event%set (weight_ref = weight)
  !!! Not implemented yet:
```

```
!      if (scale > 0)  call event%set_scales (scale)
!      if (alpha_qcd > 0)  call event%set_alpha_qcd (alpha_qcd)
   end subroutine hepeup_to_event
```

Fill the HEPEVT (event) common block:

⟨*HEP events: public*⟩+≡
```
    public :: hepevt_from_event
```

⟨*HEP events: procedures*⟩+≡
```
    subroutine hepevt_from_event (event, i_evt, keep_beams)
      type(event_t), intent(in), target :: event
      integer, intent(in), optional :: i_evt
      logical, intent(in), optional :: keep_beams
      type(particle_set_t), pointer :: particle_set
      if (event%has_particle_set ()) then
         particle_set => event%get_particle_set_ptr ()
         call hepevt_from_particle_set (particle_set, keep_beams)
         if (event%weight_prc_is_known .and. event%sqme_prc_is_known) then
            call hepevt_set_event_parameters ( &
                 weight = event%weight_prc, &
                 function_value = event%sqme_prc)
         else
            call msg_bug ("HEPEVT: event weight and/or sqme unknown")
         end if
         if (present (i_evt)) &
              call hepevt_set_event_parameters (i_evt = i_evt)
      else
         call msg_bug ("HEPEVT: event incomplete")
      end if
    end subroutine hepevt_from_event
```

19.7   LHEF Input/Output

The LHEF event record is standardized. It is an ASCII format. We try our best at using it for both input and output.

⟨`eio_lhef.f90`⟩≡
  ⟨*File header*⟩

```
  module eio_lhef

    use kinds !NODEP!
```
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
```
    use diagnostics !NODEP!
    use unit_tests

    use xml
    use lorentz !NODEP!
    use models
    use particles
    use beams
```

```
    use processes
    use events
    use eio_data
    use eio_base
    use hep_common
    use hep_events
```

⟨*Standard module head*⟩

⟨*EIO LHEF: public*⟩

⟨*EIO LHEF: types*⟩

⟨*EIO LHEF: interfaces*⟩

```
contains
```

⟨*EIO LHEF: procedures*⟩

⟨*EIO LHEF: tests*⟩

```
    end module eio_lhef
```

19.7.1   Type

With sufficient confidence that it will always be three characters, we can store
the version string with a default value.

⟨*EIO LHEF: public*⟩≡
```
    public :: eio_lhef_t
```

⟨*EIO LHEF: types*⟩≡
```
    type, extends (eio_t) :: eio_lhef_t
       logical :: writing = .false.
       logical :: reading = .false.
       integer :: unit = 0
       type(event_sample_data_t) :: data
       type(cstream_t) :: cstream
       character(3) :: version = "1.0"
       logical :: keep_beams = .false.
       logical :: recover_beams = .true.
       logical :: unweighted = .true.
       logical :: write_sqme_ref = .false.
       logical :: write_sqme_prc = .false.
       logical :: write_sqme_alt = .false.
       integer :: n_alt = 0
       integer, dimension(:), allocatable :: proc_num_id
       integer :: i_weight_sqme = 0
       type(xml_tag_t) :: tag_lhef, tag_head, tag_init, tag_event
       type(xml_tag_t), allocatable :: tag_gen_n, tag_gen_v
       type(xml_tag_t), allocatable :: tag_generator, tag_xsecinfo
       type(xml_tag_t), allocatable :: tag_sqme_ref, tag_sqme_prc
       type(xml_tag_t), dimension(:), allocatable :: tag_sqme_alt, tag_wgts_alt
       type(xml_tag_t), allocatable :: tag_weight, tag_weightinfo, tag_weights
     contains
```

⟨*EIO LHEF: eio lhef: TBP*⟩
```
    end type eio_lhef_t
```

19.7.2   Specific Methods

Set parameters that are specifically used with LHEF.

⟨*EIO LHEF: eio lhef: TBP*⟩≡
```
    procedure :: set_parameters => eio_lhef_set_parameters
```

⟨*EIO LHEF: procedures*⟩≡
```
    subroutine eio_lhef_set_parameters (eio, keep_beams, recover_beams, version, &
        extension, &
        write_sqme_ref, write_sqme_prc, write_sqme_alt)
      class(eio_lhef_t), intent(inout) :: eio
      logical, intent(in), optional :: keep_beams
      logical, intent(in), optional :: recover_beams
      character(*), intent(in), optional :: version
      type(string_t), intent(in), optional :: extension
      logical, intent(in), optional :: write_sqme_ref
      logical, intent(in), optional :: write_sqme_prc
      logical, intent(in), optional :: write_sqme_alt
      if (present (keep_beams))  eio%keep_beams = keep_beams
      if (present (recover_beams))  eio%recover_beams = recover_beams
      if (present (version)) then
         select case (version)
         case ("1.0", "2.0", "3.0")
            eio%version = version
         case default
            call msg_error ("LHEF version " // version &
                 // " is not supported.  Inserting 2.0")
            eio%version = "2.0"
         end select
      end if
      if (present (extension)) then
         eio%extension = extension
      else
         eio%extension = "lhe"
      end if
      if (present (write_sqme_ref))  eio%write_sqme_ref = write_sqme_ref
      if (present (write_sqme_prc))  eio%write_sqme_prc = write_sqme_prc
      if (present (write_sqme_alt))  eio%write_sqme_alt = write_sqme_alt
    end subroutine eio_lhef_set_parameters
```

19.7.3   Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: write => eio_lhef_write
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_write (object, unit)
      class(eio_lhef_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "LHEF event stream:"
      if (object%writing) then
         write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
      else if (object%reading) then
         write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
      else
         write (u, "(3x,A)")  "[closed]"
      end if
      write (u, "(3x,A,L1)")    "Keep beams        = ", object%keep_beams
      write (u, "(3x,A,L1)")    "Recover beams     = ", object%recover_beams
      write (u, "(3x,A,A)")     "Version           = ", object%version
      write (u, "(3x,A,A,A)")    "File extension    = '", &
           char (object%extension), "'"
      if (allocated (object%proc_num_id)) then
         write (u, "(3x,A)")  "Numerical process IDs:"
         do i = 1, size (object%proc_num_id)
            write (u, "(5x,I0,': ',I0)")  i, object%proc_num_id(i)
         end do
      end if
    end subroutine eio_lhef_write
```

Finalizer: close any open file.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: final => eio_lhef_final
```
⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_final (object)
      class(eio_lhef_t), intent(inout) :: object
      if (allocated (object%proc_num_id))  deallocate (object%proc_num_id)
      if (object%writing) then
         write (msg_buffer, "(A,A,A)")  "Events: closing LHEF file '", &
              char (object%filename), "'"
         call msg_message ()
         call object%write_footer ()
         close (object%unit)
         object%writing = .false.
      else if (object%reading) then
         write (msg_buffer, "(A,A,A)")  "Events: closing LHEF file '", &
              char (object%filename), "'"
         call msg_message ()
         call object%cstream%final ()
         object%reading = .false.
      end if
    end subroutine eio_lhef_final
```

Common initialization for input and output.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: common_init => eio_lhef_common_init
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_common_init (eio, sample, data, extension)
      class(eio_lhef_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(event_sample_data_t), intent(in), optional :: data
      if (.not. present (data)) &
           call msg_bug ("LHEF initialization: missing data")
      eio%data = data
      if (data%n_beam /= 2) &
           call msg_fatal ("LHEF: defined for scattering processes only")
      eio%unweighted = data%unweighted
      if (eio%unweighted) then
         select case (data%norm_mode)
         case (NORM_UNIT)
         case default;  call msg_fatal &
              ("LHEF: normalization for unweighted events must be '1'")
         end select
      else
         select case (data%norm_mode)
         case (NORM_SIGMA)
         case default;  call msg_fatal &
              ("LHEF: normalization for weighted events must be 'sigma'")
         end select
      end if
      eio%n_alt = data%n_alt
      eio%sample = sample
      if (present (extension)) then
         eio%extension = extension
      else
         eio%extension = "lhe"
      end if
      call eio%set_filename ()
      eio%unit = free_unit ()
      call eio%init_tags (data)
      allocate (eio%proc_num_id (data%n_proc), source = data%proc_num_id)
    end subroutine eio_lhef_common_init
```

Initialize the tag objects. Some tags depend on the LHEF version. In particular, the tags that in LHEF 2.0 identify individual weights by name in each event block, in LHEF 3.0 are replaced by info tags in the init block and a single `weights` tag in the event block. The name attributes of those tags are specific for WHIZARD.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: init_tags => eio_lhef_init_tags
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_init_tags (eio, data)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_sample_data_t), intent(in) :: data
      real(default), parameter :: pb_per_fb = 1.e-3_default
      integer :: i
      call eio%tag_lhef%init ( &
           var_str ("LesHouchesEvents"), &
```

```
      [xml_attribute (var_str ("version"), var_str (eio%version))], &
      .true.)
call eio%tag_head%init ( &
      var_str ("header"), &
      .true.)
call eio%tag_init%init ( &
      var_str ("init"), &
      .true.)
call eio%tag_event%init (var_str ("event"), &
      .true.)
select case (eio%version)
case ("1.0")
   allocate (eio%tag_gen_n)
   call eio%tag_gen_n%init ( &
        var_str ("generator_name"), &
        .true.)
   allocate (eio%tag_gen_v)
   call eio%tag_gen_v%init ( &
        var_str ("generator_version"), &
        .true.)
end select
select case (eio%version)
case ("2.0", "3.0")
   allocate (eio%tag_generator)
   call eio%tag_generator%init ( &
        var_str ("generator"), &
        [xml_attribute (var_str ("version"), var_str ("⟨Version⟩"))], &
        .true.)
   allocate (eio%tag_xsecinfo)
   call eio%tag_xsecinfo%init ( &
        var_str ("xsecinfo"), &
        [xml_attribute (var_str ("neve"), str (data%n_evt)), &
         xml_attribute (var_str ("totxsec"), &
                        str (data%total_cross_section * pb_per_fb))])
end select
select case (eio%version)
case ("2.0")
   allocate (eio%tag_weight)
   call eio%tag_weight%init (var_str ("weight"), &
        [xml_attribute (var_str ("name"))])
   if (eio%write_sqme_ref) then
      allocate (eio%tag_sqme_ref)
      call eio%tag_sqme_ref%init (var_str ("weight"), &
           [xml_attribute (var_str ("name"), var_str ("sqme_ref"))], &
           .true.)
   end if
   if (eio%write_sqme_prc) then
      allocate (eio%tag_sqme_prc)
      call eio%tag_sqme_prc%init (var_str ("weight"), &
           [xml_attribute (var_str ("name"), var_str ("sqme_prc"))], &
           .true.)
   end if
   if (eio%n_alt > 0) then
      if (eio%write_sqme_alt) then
```

```
            allocate (eio%tag_sqme_alt (1))
            call eio%tag_sqme_alt(1)%init (var_str ("weight"), &
                [xml_attribute (var_str ("name"), var_str ("sqme_alt"))], &
                .true.)
          end if
          allocate (eio%tag_wgts_alt (1))
          call eio%tag_wgts_alt(1)%init (var_str ("weight"), &
              [xml_attribute (var_str ("name"), var_str ("wgts_alt"))], &
              .true.)
        end if
    case ("3.0")
       if (eio%write_sqme_ref) then
          allocate (eio%tag_sqme_ref)
          call eio%tag_sqme_ref%init (var_str ("weightinfo"), &
              [xml_attribute (var_str ("name"), var_str ("sqme_ref"))])
       end if
       if (eio%write_sqme_prc) then
          allocate (eio%tag_sqme_prc)
          call eio%tag_sqme_prc%init (var_str ("weightinfo"), &
              [xml_attribute (var_str ("name"), var_str ("sqme_prc"))])
       end if
       if (eio%n_alt > 0) then
          if (eio%write_sqme_alt) then
             allocate (eio%tag_sqme_alt (eio%n_alt))
             do i = 1, eio%n_alt
               call eio%tag_sqme_alt(i)%init (var_str ("weightinfo"), &
                     [xml_attribute (var_str ("name"), &
                                     var_str ("sqme_alt") // str (i))])
             end do
          end if
          allocate (eio%tag_wgts_alt (eio%n_alt))
          do i = 1, eio%n_alt
             call eio%tag_wgts_alt(i)%init (var_str ("weightinfo"), &
                 [xml_attribute (var_str ("name"), &
                                 var_str ("wgts_alt") // str (i))])
          end do
       end if
       allocate (eio%tag_weightinfo)
       call eio%tag_weightinfo%init (var_str ("weightinfo"), &
           [xml_attribute (var_str ("name"))])
       allocate (eio%tag_weights)
       call eio%tag_weights%init (var_str ("weights"), .true.)
    end select
  end subroutine eio_lhef_init_tags
```

Initialize event writing.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: init_out => eio_lhef_init_out
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_init_out (eio, sample, process_ptr, data, success)
      class(eio_lhef_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
```

```
      type(event_sample_data_t), intent(in), optional :: data
      logical, intent(out), optional :: success
      integer :: u, i
      if (.not. present (data)) &
           call msg_bug ("LHEF initialization: missing data")
      eio%sample = sample
      call eio%set_splitting (data)
      call eio%common_init (sample, data)
      write (msg_buffer, "(A,A,A)")  "Events: writing to LHEF file '", &
           char (eio%filename), "'"
      call msg_message ()
      eio%writing = .true.
      u = eio%unit
      open (u, file = char (eio%filename), &
           action = "write", status = "replace")
      call eio%write_header ()
      call heprup_init &
           (data%pdg_beam, &
            data%energy_beam, &
            n_processes = data%n_proc, &
            unweighted = data%unweighted, &
            negative_weights = data%negative_weights)
      do i = 1, data%n_proc
         call heprup_set_process_parameters (i = i, &
              process_id = data%proc_num_id(i), &
              cross_section = data%cross_section(i), &
              error = data%error(i))
      end do
      call eio%tag_init%write (u);  write (u, *)
      call heprup_write_lhef (u)
      select case (eio%version)
      case ("2.0");  call eio%write_init_20 (data)
      case ("3.0");  call eio%write_init_30 (data)
      end select
      call eio%tag_init%close (u);  write (u, *)
      if (present (success))  success = .true.
    end subroutine eio_lhef_init_out
```

Initialize event reading. First read the LHEF tag and version, then read the
header and skip over its contents, then read the init block. (We require the
opening and closing tags of the init block to be placed on separate lines without
extra stuff.)

For input, we do not (yet?) support split event files.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: init_in => eio_lhef_init_in
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_init_in &
        (eio, sample, process_ptr, data, success, extension)
      class(eio_lhef_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(inout), optional :: data
```

```
      logical, intent(out), optional :: success
      logical :: exist, ok, closing
      type(event_sample_data_t) :: data_file
      type(string_t) :: string
      integer :: u
      eio%split = .false.
      call eio%common_init (sample, data, extension)
      write (msg_buffer, "(A,A,A)")  "Events: reading from LHEF file '", &
           char (eio%filename), "'"
      call msg_message ()
      inquire (file = char (eio%filename), exist = exist)
      if (.not. exist)  call msg_fatal ("Events: LHEF file not found.")
      eio%reading = .true.
      u = eio%unit
      open (u, file = char (eio%filename), &
           action = "read", status = "old")
      call eio%cstream%init (u)
      call eio%read_header ()
      call eio%tag_init%read (eio%cstream, ok)
      if (.not. ok)  call err_init
      select case (eio%version)
      case ("1.0");  call eio%read_init_10 (data_file)
         call eio%tag_init%read_content (eio%cstream, string, closing)
         if (string /= "" .or. .not. closing)  call err_init
      case ("2.0");  call eio%read_init_20 (data_file)
      case ("3.0");  call eio%read_init_30 (data_file)
      end select
      call eio%merge_data (data, data_file)
      if (present (success))  success = .true.

    contains

      subroutine err_init
        call msg_fatal ("LHEF: syntax error in init tag")
      end subroutine err_init

    end subroutine eio_lhef_init_in
```

Merge event sample data: we can check the data in the file against our assumptions and set or reset parameters.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: merge_data => eio_merge_data
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_merge_data (eio, data, data_file)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_sample_data_t), intent(inout) :: data
      type(event_sample_data_t), intent(in) :: data_file
      real, parameter :: tolerance = 1000 * epsilon (1._default)
      if (data%unweighted .neqv. data_file%unweighted)  call err_weights
      if (data%negative_weights .neqv. data_file%negative_weights) &
           call err_weights
      if (data%norm_mode /= data_file%norm_mode)  call err_norm
      if (data%n_beam /= data_file%n_beam)  call err_beams
```

```
        if (any (data%pdg_beam /= data_file%pdg_beam))  call err_beams
        if (any (abs ((data%energy_beam - data_file%energy_beam)) &
             > (data%energy_beam + data_file%energy_beam) * tolerance)) &
             call err_beams
        if (data%n_proc /= data_file%n_proc)  call err_proc
        if (any (data%proc_num_id /= data_file%proc_num_id))  call err_proc
        where (data%cross_section == 0)
           data%cross_section = data_file%cross_section
           data%error = data_file%error
        end where
        data%total_cross_section = sum (data%cross_section)
        if (data_file%n_evt > 0) then
           if (data%n_evt > 0 .and. data_file%n_evt /= data%n_evt)  call err_n_evt
           data%n_evt = data_file%n_evt
        end if
      contains
        subroutine err_weights
          call msg_fatal ("LHEF: mismatch in event weight properties")
        end subroutine err_weights
        subroutine err_norm
          call msg_fatal ("LHEF: mismatch in event normalization")
        end subroutine err_norm
        subroutine err_beams
          call msg_fatal ("LHEF: mismatch in beam properties")
        end subroutine err_beams
        subroutine err_proc
          call msg_fatal ("LHEF: mismatch in process definitions")
        end subroutine err_proc
        subroutine err_n_evt
          call msg_error ("LHEF: mismatch in specified number of events (ignored)")
        end subroutine err_n_evt
      end subroutine eio_merge_data
```

Switch from input to output: reopen the file for reading.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
      procedure :: switch_inout => eio_lhef_switch_inout
```

⟨*EIO LHEF: procedures*⟩+≡
```
      subroutine eio_lhef_switch_inout (eio, success)
        class(eio_lhef_t), intent(inout) :: eio
        logical, intent(out), optional :: success
        call msg_bug ("LHEF: in-out switch not supported")
        if (present (success))  success = .false.
      end subroutine eio_lhef_switch_inout
```

Split event file: increment the counter, close the current file, open a new one. If
the file needs a header, repeat it for the new file. (We assume that the common
block contents are still intact.)

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
      procedure :: split_out => eio_lhef_split_out
```

⟨*EIO LHEF: procedures*⟩+≡
```
      subroutine eio_lhef_split_out (eio)
        class(eio_lhef_t), intent(inout) :: eio
```

```
      integer :: u
      if (eio%split) then
         eio%split_index = eio%split_index + 1
         call eio%set_filename ()
         write (msg_buffer, "(A,A,A)")  "Events: writing to LHEF file '", &
               char (eio%filename), "'"
         call msg_message ()
         call eio%write_footer ()
         u = eio%unit
         close (u)
         open (u, file = char (eio%filename), &
               action = "write", status = "replace")
         call eio%write_header ()
         call eio%tag_init%write (u);  write (u, *)
         call heprup_write_lhef (u)
         select case (eio%version)
         case ("2.0");  call eio%write_init_20 (eio%data)
         case ("3.0");  call eio%write_init_30 (eio%data)
         end select
         call eio%tag_init%close (u);  write (u, *)
      end if
    end subroutine eio_lhef_split_out
```

Output an event. Write first the event indices, then weight and squared matrix
element, then the particle set.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: output => eio_lhef_output
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_output (eio, event, i_prc, reading)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      integer, intent(in) :: i_prc
      logical, intent(in), optional :: reading
      integer :: u
      u = output_unit (eio%unit);  if (u < 0)  return
      if (eio%writing) then
         call hepeup_from_event (event, &
               process_index = eio%proc_num_id (i_prc), &
               keep_beams = eio%keep_beams)
         write (u, '(A)') "<event>"
         call hepeup_write_lhef (eio%unit)
         select case (eio%version)
         case ("2.0");  call eio%write_event_20 (event)
         case ("3.0");  call eio%write_event_30 (event)
         end select
         write (u, '(A)') "</event>"
      else
         call eio%write ()
         call msg_fatal ("LHEF file is not open for writing")
      end if
    end subroutine eio_lhef_output
```

Input an event. Upon input of `i_prc`, we can just read in the whole HEPEUP common block. These data are known to come first. The `i_prc` value can be deduced from the IDPRUP value by a table lookup.

Reading the common block bypasses the `cstream` which accesses the input unit. This is consistent with the LHEF specification. After the common-block data have been swallowed, we can resume reading from stream.

We don't catch actual I/O errors. However, we return a negative value in `iostat` if we reached the terminating `</LesHouchesEvents>` tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
procedure :: input_i_prc => eio_lhef_input_i_prc
```

⟨*EIO LHEF: procedures*⟩+≡
```
subroutine eio_lhef_input_i_prc (eio, i_prc, iostat)
  class(eio_lhef_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  integer :: i, proc_num_id
  type(string_t) :: s
  logical :: ok
  iostat = 0
  call eio%tag_lhef%read_content (eio%cstream, s, ok)
  if (ok) then
     if (s == "") then
        iostat = -1
     else
        call err_close
     end if
     return
  else
     call eio%cstream%revert_record (s)
  end if
  call eio%tag_event%read (eio%cstream, ok)
  if (.not. ok) then
     call err_evt1
     return
  end if
  call hepeup_read_lhef (eio%unit)
  call hepeup_get_event_parameters (proc_id = proc_num_id)
  i_prc = 0
  FIND_I_PRC: do i = 1, size (eio%proc_num_id)
     if (eio%proc_num_id(i) == proc_num_id) then
        i_prc = i
        exit FIND_I_PRC
     end if
  end do FIND_I_PRC
  if (i_prc == 0)  call err_index
contains
  subroutine err_close
    call msg_error ("LHEF: reading events: syntax error in closing tag")
    iostat = 1
  end subroutine
  subroutine err_evt1
    call msg_error ("LHEF: reading events: invalid event tag, &
         &aborting read")
```

2594

```
        iostat = 2
    end subroutine err_evt1
    subroutine err_index
      call msg_error ("LHEF: reading events: undefined process ID " &
            // char (str (proc_num_id)) // ", aborting read")
      iostat = 3
    end subroutine err_index
  end subroutine eio_lhef_input_i_prc
```

Since we have already read the event information from file, this input routine can transfer the common-block contents to the event record. Also, we read any further information in the event record.

Since LHEF doesn't give this information, we must assume that the MCI group, term, and channel can all be safely set to 1. This works if there is only one MCI group and term. The channel doesn't matter for the matrix element.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: input_event => eio_lhef_input_event
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_input_event (eio, event, iostat)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      type(particle_set_t), pointer :: pset
      type(string_t) :: s
      logical :: closing
      iostat = 0
      call event%reset ()
      call event%select (1, 1, 1)
      call hepeup_to_event (event, recover_beams = eio%recover_beams)
      if (associated (event%process)) then
         pset => event%get_particle_set_ptr ()
         call particle_set_set_model (pset, event%process%get_model_ptr ())
      end if
      select case (eio%version)
      case ("1.0")
         call eio%tag_event%read_content (eio%cstream, s, closing = closing)
         if (s /= "" .or. .not. closing)  call err_evt2
      case ("2.0");  call eio%read_event_20 (event)
      case ("3.0");  call eio%read_event_30 (event)
      end select
    contains
      subroutine err_evt2
        call msg_error ("LHEF: reading events: syntax error in event record, &
              &aborting read")
        iostat = 2
      end subroutine err_evt2

    end subroutine eio_lhef_input_event
```

19.7.4 Les Houches Event File: header/footer

These two routines write the header and footer for the Les Houches Event File format (LHEF).

The current version writes no information except for the generator name and version (v.1.0 only).

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: write_header => eio_lhef_write_header
procedure :: write_footer => eio_lhef_write_footer
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_write_header (eio)
  class(eio_lhef_t), intent(in) :: eio
  integer :: u
  u = output_unit (eio%unit);  if (u < 0)  return
  call eio%tag_lhef%write (u);  write (u, *)
  call eio%tag_head%write (u);  write (u, *)
  select case (eio%version)
  case ("1.0")
     write (u, "(2x)", advance = "no")
     call eio%tag_gen_n%write (var_str ("WHIZARD"), u)
     write (u, *)
     write (u, "(2x)", advance = "no")
     call eio%tag_gen_v%write (var_str ("⟨Version⟩"), u)
     write (u, *)
  end select
  call eio%tag_head%close (u);  write (u, *)
end subroutine eio_lhef_write_header

subroutine eio_lhef_write_footer (eio)
  class(eio_lhef_t), intent(in) :: eio
  integer :: u
  u = output_unit (eio%unit);  if (u < 0)  return
  call eio%tag_lhef%close (u)
end subroutine eio_lhef_write_footer
```

Reading the header just means finding the tags and ignoring any contents. When done, we should stand just after the header tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: read_header => eio_lhef_read_header
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_read_header (eio)
  class(eio_lhef_t), intent(inout) :: eio
  logical :: success, closing
  type(string_t) :: content
  call eio%tag_lhef%read (eio%cstream, success)
  if (.not. success .or. .not. eio%tag_lhef%has_content)  call err_lhef
  if (eio%tag_lhef%get_attribute (1) /= eio%version)  call err_version
  call eio%tag_head%read (eio%cstream, success)
  if (.not. success)  call err_header
  if (eio%tag_head%has_content) then
     SKIP_HEADER_CONTENT: do
        call eio%tag_head%read_content (eio%cstream, content, closing)
```

```
          if (closing)  exit SKIP_HEADER_CONTENT
        end do SKIP_HEADER_CONTENT
    end if
  contains
    subroutine err_lhef
      call msg_fatal ("LHEF: LesHouchesEvents tag absent or corrupted")
    end subroutine err_lhef
    subroutine err_header
      call msg_fatal ("LHEF: header tag absent or corrupted")
    end subroutine err_header
    subroutine err_version
      call msg_error ("LHEF: version mismatch: expected " &
           // eio%version // ", found " &
           // char (eio%tag_lhef%get_attribute (1)))
    end subroutine err_version
  end subroutine eio_lhef_read_header
```

19.7.5   Version-Specific Code: 1.0

In version 1.0, the init tag contains just HEPRUP data. While a `cstream` is
connected to the input unit, we bypass it temporarily for the purpose of reading
the HEPRUP contents. This is consistent with the LHEF standard.

This routine does not read the closing tag of the init block.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: read_init_10 => eio_lhef_read_init_10
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_read_init_10 (eio, data)
      class(eio_lhef_t), intent(in) :: eio
      type(event_sample_data_t), intent(out) :: data
      integer :: n_proc, i
      call heprup_read_lhef (eio%unit)
      call heprup_get_run_parameters (n_processes = n_proc)
      call data%init (n_proc)
      data%n_beam = 2
      call heprup_get_run_parameters ( &
           unweighted = data%unweighted, &
           negative_weights = data%negative_weights, &
           beam_pdg = data%pdg_beam, &
           beam_energy = data%energy_beam)
      if (data%unweighted) then
         data%norm_mode = NORM_UNIT
      else
         data%norm_mode = NORM_SIGMA
      end if
      do i = 1, n_proc
         call heprup_get_process_parameters (i, &
              process_id = data%proc_num_id(i), &
              cross_section = data%cross_section(i), &
              error = data%error(i))
      end do
    end subroutine eio_lhef_read_init_10
```

19.7.6 Version-Specific Code: 2.0

This is the init information for the 2.0 format, after the HEPRUP data. We have the following tags:

- `generator` Generator name and version.

- `xsecinfo` Cross section and weights data. We have the total cross section and number of events (assuming that the event file is intact), but information on minimum and maximum weights is not available before the file is complete. We just write the mandatory tags. (Note that the default values of the other tags describe a uniform unit weight, but we can determine most values only after the sample is complete.)

- `cutsinfo` This optional tag is too specific to represent the possibilities of WHIZARD, so we skip it.

- `procinfo` This optional tag is useful for giving details of NLO calculations. Skipped.

- `mergetype` Optional, also not applicable.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: write_init_20 => eio_lhef_write_init_20
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_write_init_20 (eio, data)
      class(eio_lhef_t), intent(in) :: eio
      type(event_sample_data_t), intent(in) :: data
      integer :: u
      u = eio%unit
      call eio%tag_generator%write (u)
      write (u, "(A)", advance="no")  "WHIZARD"
      call eiO%tag_generator%close (u);  write (u, *)
      call eio%tag_xsecinfo%write (u);  write (u, *)
    end subroutine eio_lhef_write_init_20
```

When reading the init block, we first call the 1.0 routine that fills HEPRUP. Then we consider the possible tags. Only the `generator` and `xsecinfo` tags are of interest. We skip everything else except for the closing tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: read_init_20 => eio_lhef_read_init_20
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_read_init_20 (eio, data)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_sample_data_t), intent(out) :: data
      real(default), parameter :: pb_per_fb = 1.e-3_default
      type(string_t) :: content
      logical :: found, closing
      call eio_lhef_read_init_10 (eio, data)
      SCAN_INIT_TAGS: do
         call eio%tag_generator%read (eio%cstream, found)
         if (found) then
            if (.not. eio%tag_generator%has_content)  call err_generator
```

```
            call eio%tag_generator%read_content (eio%cstream, content, closing)
            call msg_message ("LHEF: Event file has been generated by " &
                 // char (content) // " " &
                 // char (eio%tag_generator%get_attribute (1)))
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_xsecinfo%read (eio%cstream, found)
         if (found) then
            if (eio%tag_xsecinfo%has_content)  call err_xsecinfo
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_init%read_content (eio%cstream, content, closing)
         if (closing) then
            if (content /= "")  call err_init
            exit SCAN_INIT_TAGS
         end if
      end do SCAN_INIT_TAGS
      data%n_evt = &
           read_ival (eio%tag_xsecinfo%get_attribute (1))
      data%total_cross_section = &
           read_rval (eio%tag_xsecinfo%get_attribute (2)) / pb_per_fb
    contains
      subroutine err_generator
        call msg_fatal ("LHEF: invalid generator tag")
      end subroutine err_generator
      subroutine err_xsecinfo
        call msg_fatal ("LHEF: invalid xsecinfo tag")
      end subroutine err_xsecinfo
      subroutine err_init
        call msg_fatal ("LHEF: syntax error after init tag")
      end subroutine err_init
    end subroutine eio_lhef_read_init_20
```

This is additional event-specific information for the 2.0 format, after the HEP-EUP data. We can specify weights, starting from the master weight and adding alternative weights. The alternative weights are collected in a common tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: write_event_20 => eio_lhef_write_event_20
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_write_event_20 (eio, event)
      class(eio_lhef_t), intent(in) :: eio
      type(event_t), intent(in) :: event
      type(string_t) :: s
      integer :: i, u
      u = eio%unit
      if (eio%write_sqme_ref) then
         s = str (event%sqme_ref)
         call eio%tag_sqme_ref%write (s, u);  write (u, *)
      end if
      if (eio%write_sqme_prc) then
         s = str (event%sqme_prc)
         call eio%tag_sqme_prc%write (s, u);  write (u, *)
      end if
```

```
       if (eio%n_alt > 0) then
          if (eio%write_sqme_alt) then
             s = str (event%sqme_alt(1))
             do i = 2, eio%n_alt
                s = s // " " // str (event%sqme_alt(i));  write (u, *)
             end do
             call eio%tag_sqme_alt(1)%write (s, u)
          end if
          s = str (event%weight_alt(1))
          do i = 2, eio%n_alt
             s = s // " " // str (event%weight_alt(i));  write (u, *)
          end do
          call eio%tag_wgts_alt(1)%write (s, u)
       end if
    end subroutine eio_lhef_write_event_20
```

Read extra event data. If there is a weight entry labeled `sqme_prc`, we take
this as the squared matrix-element value (the new *reference* value `sqme_ref`).
Other tags, including tags written by the above writer, are skipped.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: read_event_20 => eio_lhef_read_event_20
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_read_event_20 (eio, event)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_t), intent(inout) :: event
      type(string_t) :: content
      logical :: found, closing
      SCAN_EVENT_TAGS: do
         call eio%tag_weight%read (eio%cstream, found)
         if (found) then
            if (.not. eio%tag_weight%has_content)  call err_weight
            call eio%tag_weight%read_content (eio%cstream, content, closing)
            if (.not. closing)  call err_weight
            if (eio%tag_weight%get_attribute (1) == "sqme_prc") then
               call event%set (sqme_ref = read_rval (content))
            end if
            cycle SCAN_EVENT_TAGS
         end if
         call eio%tag_event%read_content (eio%cstream, content, closing)
         if (closing) then
            if (content /= "")  call err_event
            exit SCAN_EVENT_TAGS
         end if
      end do SCAN_EVENT_TAGS
    contains
      subroutine err_weight
        call msg_fatal ("LHEF: invalid weight tag in event record")
      end subroutine err_weight
      subroutine err_event
        call msg_fatal ("LHEF: syntax error after event tag")
      end subroutine err_event
    end subroutine eio_lhef_read_event_20
```

19.7.7 Version-Specific Code: 3.0

This is the init information for the 3.0 format, after the HEPRUP data. We have the following tags:

- `generator` Generator name and version.

- `xsecinfo` Cross section and weights data. We have the total cross section and number of events (assuming that the event file is intact), but information on minimum and maximum weights is not available before the file is complete. We just write the mandatory tags. (Note that the default values of the other tags describe a uniform unit weight, but we can determine most values only after the sample is complete.)

- `cutsinfo` This optional tag is too specific to represent the possibilities of WHIZARD, so we skip it.

- `procinfo` This optional tag is useful for giving details of NLO calculations. Skipped.

- `weightinfo` Determine the meaning of optional weights, whose values are given in the event record.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: write_init_30 => eio_lhef_write_init_30
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_write_init_30 (eio, data)
      class(eio_lhef_t), intent(in) :: eio
      type(event_sample_data_t), intent(in) :: data
      real(default), parameter :: pb_per_fb = 1.e-3_default
      integer :: u, i
      u = output_unit (eio%unit)
      call eio%tag_generator%write (u)
      write (u, "(A)", advance="no")  "WHIZARD"
      call ei0%tag_generator%close (u);  write (u, *)
      call eio%tag_xsecinfo%write (u);  write (u, *)
      if (eio%write_sqme_ref) then
         call eio%tag_sqme_ref%write (u);  write (u, *)
      end if
      if (eio%write_sqme_prc) then
         call eio%tag_sqme_prc%write (u);  write (u, *)
      end if
      if (eio%write_sqme_alt) then
         do i = 1, eio%n_alt
            call eio%tag_sqme_alt(i)%write (u);  write (u, *)
         end do
      end if
      do i = 1, eio%n_alt
         call eio%tag_wgts_alt(i)%write (u);  write (u, *)
      end do
    end subroutine eio_lhef_write_init_30
```

When reading the init block, we first call the 1.0 routine that fills HEPRUP. Then we consider the possible tags. Only the `generator` and `xsecinfo` tags are of interest. We skip everything else except for the closing tag.

⟨EIO LHEF: eio lhef: TBP⟩+≡
```
    procedure :: read_init_30 => eio_lhef_read_init_30
```

⟨EIO LHEF: procedures⟩+≡
```
    subroutine eio_lhef_read_init_30 (eio, data)
      class(eio_lhef_t), intent(inout) :: eio
      type(event_sample_data_t), intent(out) :: data
      real(default), parameter :: pb_per_fb = 1.e-3_default
      type(string_t) :: content
      logical :: found, closing
      integer :: n_weightinfo
      call eio_lhef_read_init_10 (eio, data)
      n_weightinfo = 0
      eio%i_weight_sqme = 0
      SCAN_INIT_TAGS: do
         call eio%tag_generator%read (eio%cstream, found)
         if (found) then
            if (.not. eio%tag_generator%has_content)  call err_generator
            call eio%tag_generator%read_content (eio%cstream, content, closing)
            call msg_message ("LHEF: Event file has been generated by " &
                 // char (content) // " " &
                 // char (eio%tag_generator%get_attribute (1)))
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_xsecinfo%read (eio%cstream, found)
         if (found) then
            if (eio%tag_xsecinfo%has_content)  call err_xsecinfo
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_weightinfo%read (eio%cstream, found)
         if (found) then
            if (eio%tag_weightinfo%has_content)  call err_xsecinfo
            n_weightinfo = n_weightinfo + 1
            if (eio%tag_weightinfo%get_attribute (1) == "sqme_prc") then
               eio%i_weight_sqme = n_weightinfo
            end if
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_init%read_content (eio%cstream, content, closing)
         if (closing) then
            if (content /= "")  call err_init
            exit SCAN_INIT_TAGS
         end if
      end do SCAN_INIT_TAGS
      data%n_evt = &
           read_ival (eio%tag_xsecinfo%get_attribute (1))
      data%total_cross_section = &
           read_rval (eio%tag_xsecinfo%get_attribute (2)) / pb_per_fb
    contains
      subroutine err_generator
        call msg_fatal ("LHEF: invalid generator tag")
```

```
      end subroutine err_generator
      subroutine err_xsecinfo
        call msg_fatal ("LHEF: invalid xsecinfo tag")
      end subroutine err_xsecinfo
      subroutine err_init
        call msg_fatal ("LHEF: syntax error after init tag")
      end subroutine err_init
    end subroutine eio_lhef_read_init_30
```

This is additional event-specific information for the 3.0 format, after the HEP-EUP data. We can specify weights, starting from the master weight and adding alternative weights. The weight tags are already allocated, so we just have to transfer the weight values to strings, assemble them and write them to file. All weights are collected in a single tag.

Note: If efficiency turns out to be an issue, we may revert to traditional character buffer writing. However, we need to know the maximum length.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
    procedure :: write_event_30 => eio_lhef_write_event_30
```

⟨*EIO LHEF: procedures*⟩+≡
```
    subroutine eio_lhef_write_event_30 (eio, event)
      class(eio_lhef_t), intent(in) :: eio
      type(event_t), intent(in) :: event
      type(string_t) :: s
      integer :: u, i
      u = eio%unit
      s = ""
      if (eio%write_sqme_ref) then
         s = s // str (event%sqme_ref) // " "
      end if
      if (eio%write_sqme_prc) then
         s = s // str (event%sqme_prc) // " "
      end if
      if (eio%n_alt > 0) then
         if (eio%write_sqme_alt) then
            s = s // str (event%sqme_alt(1)) // " "
            do i = 2, eio%n_alt
               s = s // str (event%sqme_alt(i)) // " "
            end do
         end if
         s = s // str (event%weight_alt(1)) // " "
         do i = 2, eio%n_alt
            s = s // str (event%weight_alt(i)) // " "
         end do
      end if
      if (len_trim (s) > 0) then
         call eio%tag_weights%write (trim (s), u);  write (u, *)
      end if
    end subroutine eio_lhef_write_event_30
```

Read extra event data. If there is a `weights` tag and if there was a `weightinfo` entry labeled `sqme_prc`, we extract the corresponding entry from the weights

string and store this as the event's squared matrix-element value. Other tags, including tags written by the above writer, are skipped.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
procedure :: read_event_30 => eio_lhef_read_event_30
```

⟨*EIO LHEF: procedures*⟩+≡
```
subroutine eio_lhef_read_event_30 (eio, event)
  class(eio_lhef_t), intent(inout) :: eio
  type(event_t), intent(inout) :: event
  type(string_t) :: content, string
  logical :: found, closing
  integer :: i
  SCAN_EVENT_TAGS: do
     call eio%tag_weights%read (eio%cstream, found)
     if (found) then
        if (.not. eio%tag_weights%has_content)  call err_weights
        call eio%tag_weights%read_content (eio%cstream, content, closing)
        if (.not. closing)  call err_weights
        if (eio%i_weight_sqme > 0) then
           SCAN_WEIGHTS: do i = 1, eio%i_weight_sqme
              call split (content, string, " ")
              content = adjustl (content)
              if (i == eio%i_weight_sqme) then
                 call event%set (sqme_ref = read_rval (string))
                 exit SCAN_WEIGHTS
              end if
           end do SCAN_WEIGHTS
        end if
        cycle SCAN_EVENT_TAGS
     end if
     call eio%tag_event%read_content (eio%cstream, content, closing)
     if (closing) then
        if (content /= "")  call err_event
        exit SCAN_EVENT_TAGS
     end if
  end do SCAN_EVENT_TAGS
contains
  subroutine err_weights
    call msg_fatal ("LHEF: invalid weights tag in event record")
  end subroutine err_weights
  subroutine err_event
    call msg_fatal ("LHEF: syntax error after event tag")
  end subroutine err_event
end subroutine eio_lhef_read_event_30
```

19.7.8 Auxiliary

Create a string from a number. We use fixed format for the reals and variable format for integers.

⟨*EIO LHEF: interfaces*⟩≡
```
interface str
   module procedure str_int, str_real
end interface
```

⟨*EIO LHEF: procedures*⟩+≡

```
function str_int (i) result (s)
  integer, intent(in) :: i
  type(string_t) :: s
  character(32) :: buffer
  write (buffer, "(I0)")  i
  s = var_str (trim (buffer))
end function str_int

function str_real (x) result (s)
  real(default), intent(in) :: x
  type(string_t) :: s
  character(32) :: buffer
  write (buffer, "(ES17.10)")  x
  s = var_str (trim (adjustl (buffer)))
end function str_real
```

Auxiliary: Read real, integer, string value.

⟨*EIO LHEF: procedures*⟩+≡

```
function read_rval (s) result (rval)
  type(string_t), intent(in) :: s
  real(default) :: rval
  character(80) :: buffer
  buffer = s
  read (buffer, *)  rval
end function read_rval

function read_ival (s) result (ival)
  type(string_t), intent(in) :: s
  integer :: ival
  character(80) :: buffer
  buffer = s
  read (buffer, *)  ival
end function read_ival
```

19.7.9 Unit tests

⟨*EIO LHEF: public*⟩+≡

```
public :: eio_lhef_test
```

⟨*EIO LHEF: tests*⟩≡

```
subroutine eio_lhef_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
  ⟨EIO LHEF: execute tests⟩
end subroutine eio_lhef_test
```

Version 1.0 Output

We test the implementation of all I/O methods. We start with output according to version 1.0.

⟨*EIO LHEF: execute tests*⟩≡
```
    call test (eio_lhef_1, "eio_lhef_1", &
         "write version 1.0", &
         u, results)
```
⟨*EIO LHEF: tests*⟩+≡
```
    subroutine eio_lhef_1 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_lhef_1"
      write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
      process_ptr%ptr => process
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%unweighted = .true.
      data%norm_mode = NORM_UNIT
      data%pdg_beam = 25
      data%energy_beam = 500
      data%proc_num_id = [42]
      data%cross_section(1) = 100
      data%error(1) = 1
      data%total_cross_section = sum (data%cross_section)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_lhef_1"
```

```
allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters ()
end select

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // "." // eio%extension), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters ()
end select

select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)
```

```
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_lhef_1"

   end subroutine eio_lhef_1
```

Version 2.0 Output

Version 2.0 has added a lot of options to the LHEF format. We implement some
of them.

⟨*EIO LHEF: execute tests*⟩+≡
```
      call test (eio_lhef_2, "eio_lhef_2", &
           "write version 2.0", &
           u, results)
```

⟨*EIO LHEF: tests*⟩+≡
```
   subroutine eio_lhef_2 (u)
     integer, intent(in) :: u
     type(model_list_t) :: model_list
     type(event_t), allocatable, target :: event
     type(process_t), allocatable, target :: process
     type(process_ptr_t) :: process_ptr
     type(process_instance_t), allocatable, target :: process_instance
     type(event_sample_data_t) :: data
     class(eio_t), allocatable :: eio
     type(string_t) :: sample
     integer :: u_file, iostat
     character(80) :: buffer

     write (u, "(A)")  "* Test output: eio_lhef_2"
     write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
     write (u, "(A)")

     call syntax_model_file_init ()

     write (u, "(A)")  "* Initialize test process"

     allocate (process)
     process_ptr%ptr => process
     allocate (process_instance)
     call prepare_test_process (process, process_instance, model_list)
     call process_instance%setup_event_data ()

     allocate (event)
     call event%basic_init ()
     call event%connect (process_instance, process%get_model_ptr ())

     call data%init (1)
     data%unweighted = .false.
     data%norm_mode = NORM_SIGMA
```

```
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_lhef_2"

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (version = "2.0", write_sqme_prc = .true.)
end select

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])


call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // "." // eio%extension), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:10) == "<generator")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)
```

```
      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_lhef_2"

   end subroutine eio_lhef_2
```

Version 3.0 Output

Version 3.0 is an update which removes some tags (which we didn't use anyway)
and suggests a new treatment of weights.

⟨*EIO LHEF: execute tests*⟩+≡
```
      call test (eio_lhef_3, "eio_lhef_3", &
           "write version 3.0", &
           u, results)
```

⟨*EIO LHEF: tests*⟩+≡
```
   subroutine eio_lhef_3 (u)
     integer, intent(in) :: u
     type(model_list_t) :: model_list
     type(event_t), allocatable, target :: event
     type(process_t), allocatable, target :: process
     type(process_ptr_t) :: process_ptr
     type(process_instance_t), allocatable, target :: process_instance
     type(event_sample_data_t) :: data
     class(eio_t), allocatable :: eio
     type(string_t) :: sample
     integer :: u_file, iostat
     character(80) :: buffer

     write (u, "(A)")  "* Test output: eio_lhef_3"
     write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
     write (u, "(A)")

     call syntax_model_file_init ()

     write (u, "(A)")  "* Initialize test process"

     allocate (process)
     process_ptr%ptr => process
     allocate (process_instance)
     call prepare_test_process (process, process_instance, model_list)
     call process_instance%setup_event_data ()

     allocate (event)
     call event%basic_init ()
     call event%connect (process_instance, process%get_model_ptr ())

     call data%init (1)
     data%unweighted = .false.
     data%norm_mode = NORM_SIGMA
     data%n_evt = 1
```

2610

```
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_lhef_3"

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (version = "3.0", write_sqme_prc = .true.)
end select

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])


call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".lhe"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:10) == "<generator")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

call model_list%final ()
```

```
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_lhef_3"

    end subroutine eio_lhef_3
```

Version 1.0 Input

Check input of a version-1.0 conforming LHEF file.

⟨*EIO LHEF: execute tests*⟩+≡
```
      call test (eio_lhef_4, "eio_lhef_4", &
           "read version 1.0", &
           u, results)
```

⟨*EIO LHEF: tests*⟩+≡
```
    subroutine eio_lhef_4 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat, i_prc

      write (u, "(A)")  "* Test output: eio_lhef_4"
      write (u, "(A)")  "*   Purpose: read a LHEF 1.0 file"
      write (u, "(A)")

      write (u, "(A)")  "* Write a LHEF data file"
      write (u, "(A)")

      u_file = free_unit ()
      sample = "eio_lhef_4"
      open (u_file, file = char (sample // ".lhe"), &
           status = "replace", action = "readwrite")

      write (u_file, "(A)")  '<LesHouchesEvents version="1.0">'
      write (u_file, "(A)")  '<header>'
      write (u_file, "(A)")  '  <arbitrary_tag opt="foo">content</arbitrary_tag>'
      write (u_file, "(A)")  '  Text'
      write (u_file, "(A)")  '  <another_tag />'
      write (u_file, "(A)")  '</header>'
      write (u_file, "(A)")  '<init>'
      write (u_file, "(A)")  ' 25 25  5.0000000000E+02  5.0000000000E+02 &
           & -1 -1 -1 -1 3 1'
      write (u_file, "(A)")  '  1.0000000000E-01  1.0000000000E-03 &
           & 1.0000000000E+00 42'
      write (u_file, "(A)")  '</init>'
      write (u_file, "(A)")  '<event>'
```

```
write (u_file, "(A)")  ' 4 42  3.0574068604E+08  1.0000000000E+03 &
     & -1.0000000000E+00 -1.0000000000E+00'
write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00  0.0000000000E+00 &
     & 4.8412291828E+02  5.0000000000E+02  1.2500000000E+02 &
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00  0.0000000000E+00 &
     &-4.8412291828E+02  5.0000000000E+02  1.2500000000E+02 &
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0 -1.4960220911E+02 -4.6042825611E+02 &
     & 0.0000000000E+00  5.0000000000E+02  1.2500000000E+02 &
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0  1.4960220911E+02  4.6042825611E+02 &
     & 0.0000000000E+00  5.0000000000E+02  1.2500000000E+02 &
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  '</event>'
write (u_file, "(A)")  '</LesHouchesEvents>'
close (u_file)


write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call syntax_model_file_init ()

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (recover_beams = .false.)
end select

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%norm_mode = NORM_UNIT
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize and read header"
write (u, "(A)")
```

```
    call eio%init_in (sample, [process_ptr], data)
    call eio%write (u)

    write (u, *)

    select type (eio)
    type is (eio_lhef_t)
       call eio%tag_lhef%write (u);  write (u, *)
    end select

    write (u, *)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read event"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)

    select type (eio)
    type is (eio_lhef_t)
       write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
            " with ID = ", eio%proc_num_id(i_prc)
    end select

    call eio%input_event (event, iostat)

    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read closing"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)
    write (u, "(A,I0)")  "iostat = ", iostat

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()
    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lhef_4"

end subroutine eio_lhef_4
```

Version 2.0 Input

Check input of a version-2.0 conforming LHEF file.

⟨*EIO LHEF: execute tests*⟩+≡
```
call test (eio_lhef_5, "eio_lhef_5", &
    "read version 2.0", &
    u, results)
```

⟨*EIO LHEF: tests*⟩+≡
```
subroutine eio_lhef_5 (u)
  integer, intent(in) :: u
  type(model_list_t) :: model_list
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_ptr_t) :: process_ptr
  type(process_instance_t), allocatable, target :: process_instance
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat, i_prc

  write (u, "(A)")  "* Test output: eio_lhef_5"
  write (u, "(A)")  "*   Purpose: read a LHEF 2.0 file"
  write (u, "(A)")

  write (u, "(A)")  "* Write a LHEF data file"
  write (u, "(A)")

  u_file = free_unit ()
  sample = "eio_lhef_5"
  open (u_file, file = char (sample // ".lhe"), &
      status = "replace", action = "readwrite")

  write (u_file, "(A)")  '<LesHouchesEvents version="2.0">'
  write (u_file, "(A)")  '<header>'
  write (u_file, "(A)")  '</header>'
  write (u_file, "(A)")  '<init>'
  write (u_file, "(A)")  ' 25 25  5.0000000000E+02  5.0000000000E+02 &
      &-1 -1 -1 -1 4 1'
  write (u_file, "(A)")  '  1.0000000000E-01  1.0000000000E-03 &
      & 0.0000000000E+00 42'
  write (u_file, "(A)")  '<generator version="2.2.0">WHIZARD&
      &</generator>'
  write (u_file, "(A)")  '<xsecinfo neve="1" totxsec="1.0000000000E-01" />'
  write (u_file, "(A)")  '</init>'
  write (u_file, "(A)")  '<event>'
  write (u_file, "(A)")  ' 4 42  3.0574068604E+08  1.0000000000E+03 &
      &-1.0000000000E+00 -1.0000000000E+00'
  write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
      & 0.0000000000E+00  4.8412291828E+02  5.0000000000E+02 &
      & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
  write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
```

```
       & 0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02 &
       & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0 -1.4960220911E+02 &
       &-4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
       & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0  1.4960220911E+02 &
       & 4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
       & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  '<weight name="sqme_prc">1.0000000000E+00</weight>'
write (u_file, "(A)")  '</event>'
write (u_file, "(A)")  '</LesHouchesEvents>'
close (u_file)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call syntax_model_file_init ()

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (version = "2.0", recover_beams = .false.)
end select

call data%init (1)
data%unweighted = .false.
data%norm_mode = NORM_SIGMA
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize and read header"
write (u, "(A)")

call eio%init_in (sample, [process_ptr], data)
call eio%write (u)

write (u, *)

select type (eio)
type is (eio_lhef_t)
```

2616

```
      call eio%tag_lhef%write (u);  write (u, *)
   end select

   write (u, *)
   call data%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Read event"
   write (u, "(A)")

   call eio%input_i_prc (i_prc, iostat)

   select type (eio)
   type is (eio_lhef_t)
      write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
           " with ID = ", eio%proc_num_id(i_prc)
   end select

   call eio%input_event (event, iostat)

   call event%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Read closing"
   write (u, "(A)")

   call eio%input_i_prc (i_prc, iostat)
   write (u, "(A,I0)")  "iostat = ", iostat

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call eio%final ()
   call event%final ()
   deallocate (event)

   call cleanup_test_process (process, process_instance)
   deallocate (process_instance)
   deallocate (process)

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: eio_lhef_5"

 end subroutine eio_lhef_5
```

Version 3.0 Input

Check input of a version-3.0 conforming LHEF file.

⟨*EIO LHEF: execute tests*⟩+≡

```
          call test (eio_lhef_6, "eio_lhef_6", &
               "read version 3.0", &
               u, results)
```

⟨*EIO LHEF: tests*⟩+≡

```
     subroutine eio_lhef_6 (u)
       integer, intent(in) :: u
       type(model_list_t) :: model_list
       type(event_t), allocatable, target :: event
       type(process_t), allocatable, target :: process
       type(process_ptr_t) :: process_ptr
       type(process_instance_t), allocatable, target :: process_instance
       type(event_sample_data_t) :: data
       class(eio_t), allocatable :: eio
       type(string_t) :: sample
       integer :: u_file, iostat, i_prc

       write (u, "(A)")  "* Test output: eio_lhef_6"
       write (u, "(A)")  "*   Purpose: read a LHEF 3.0 file"
       write (u, "(A)")

       write (u, "(A)")  "* Write a LHEF data file"
       write (u, "(A)")

       u_file = free_unit ()
       sample = "eio_lhef_6"
       open (u_file, file = char (sample // ".lhe"), &
             status = "replace", action = "readwrite")

       write (u_file, "(A)")  '<LesHouchesEvents version="3.0">'
       write (u_file, "(A)")  '<header>'
       write (u_file, "(A)")  '</header>'
       write (u_file, "(A)")  '<init>'
       write (u_file, "(A)")  ' 25 25  5.0000000000E+02  5.0000000000E+02 &
            &-1 -1 -1 -1 4 1'
       write (u_file, "(A)")  '  1.0000000000E-01  1.0000000000E-03 &
            & 0.0000000000E+00 42'
       write (u_file, "(A)")  '<generator version="2.2.0">WHIZARD&
            &</generator>'
       write (u_file, "(A)")  '<xsecinfo neve="1" totxsec="1.0000000000E-01" />'
       write (u_file, "(A)")  '<weightinfo name="sqme_prc" />'
       write (u_file, "(A)")  '</init>'
       write (u_file, "(A)")  '<event>'
       write (u_file, "(A)")  ' 4 42  3.0574068604E+08  1.0000000000E+03 &
            &-1.0000000000E+00 -1.0000000000E+00'
       write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
            & 0.0000000000E+00  4.8412291828E+02  5.0000000000E+02 &
            & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
       write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
            & 0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02 &
            & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
       write (u_file, "(A)")  ' 25 1 1 2 0 0 -1.4960220911E+02 &
            &-4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
            & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
       write (u_file, "(A)")  ' 25 1 1 2 0 0  1.4960220911E+02 &
```

```
      & 4.6042825611E+02   0.0000000000E+00   5.0000000000E+02 &
      & 1.2500000000E+02   0.0000000000E+00   9.0000000000E+00'
write (u_file, "(A)")  '<weights>1.0000000000E+00</weights>'
write (u_file, "(A)")  '</event>'
write (u_file, "(A)")  '</LesHouchesEvents>'
close (u_file)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call syntax_model_file_init ()

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (version = "3.0", recover_beams = .false.)
end select

call data%init (1)
data%unweighted = .false.
data%norm_mode = NORM_SIGMA
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize and read header"
write (u, "(A)")

call eio%init_in (sample, [process_ptr], data)
call eio%write (u)

write (u, *)

select type (eio)
type is (eio_lhef_t)
   call eio%tag_lhef%write (u);  write (u, *)
end select

write (u, *)
call data%write (u)
```

```
      write (u, "(A)")
      write (u, "(A)")  "* Read event"
      write (u, "(A)")

      call eio%input_i_prc (i_prc, iostat)

      select type (eio)
      type is (eio_lhef_t)
         write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
               " with ID = ", eio%proc_num_id(i_prc)
      end select

      call eio%input_event (event, iostat)

      call event%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Read closing"
      write (u, "(A)")

      call eio%input_i_prc (i_prc, iostat)
      write (u, "(A,I0)") "iostat = ", iostat

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call eio%final ()
      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_lhef_6"

  end subroutine eio_lhef_6
```

19.8   HepMC events

This section provides the interface to the HepMC C++ library for handling Monte-Carlo events.

Each C++ class of HepMC that we use is mirrored by a Fortran type, which contains as its only component the C pointer to the C++ object.

Each C++ method of HepMC that we use has a C wrapper function. This function takes a pointer to the host object as its first argument. Further arguments are either C pointers, or in the case of simple types (integer, real),

interoperable C/Fortran objects.

The C wrapper functions have explicit interfaces in the Fortran module. They are called by Fortran wrapper procedures. These are treated as methods of the corresponding Fortran type.

⟨hepmc_interface.f90⟩≡
  ⟨*File header*⟩

  module hepmc_interface

    use iso_c_binding !NODEP!
  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use constants !NODEP!
    use lorentz !NODEP!
    use unit_tests
    use models
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use polarizations

  ⟨*Standard module head*⟩

  ⟨*HepMC interface: public*⟩

  ⟨*HepMC interface: types*⟩

  ⟨*HepMC interface: interfaces*⟩

  contains

  ⟨*HepMC interface: procedures*⟩

  ⟨*HepMC interface: tests*⟩

  end module hepmc_interface

19.8.1   Interface check

This function can be called in order to verify that we are using the actual HepMC library, and not the dummy version.

⟨*HepMC interface: interfaces*⟩≡
    interface
       logical(c_bool) function hepmc_available () bind(C)
         import
       end function hepmc_available
    end interface

⟨*HepMC interface: public*⟩≡
    public :: hepmc_is_available

⟨*HepMC interface: procedures*⟩≡
```
function hepmc_is_available () result (flag)
  logical :: flag
  flag = hepmc_available ()
end function hepmc_is_available
```

19.8.2 FourVector

The C version of four-vectors is often transferred by value, and the associated procedures are all inlined. The wrapper needs to transfer by reference, so we create FourVector objects on the heap which have to be deleted explicitly. The input is a `vector4_t` or `vector3_t` object from the `lorentz` module.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_four_vector_t
```

⟨*HepMC interface: types*⟩≡
```
type :: hepmc_four_vector_t
  private
  type(c_ptr) :: obj
end type hepmc_four_vector_t
```

In the C constructor, the zero-component (fourth argument) is optional; if missing, it is set to zero. The Fortran version has initializer form and takes either a three-vector or a four-vector. A further version extracts the four-vector from a HepMC particle object.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
  type(c_ptr) function new_four_vector_xyz (x, y, z) bind(C)
    import
    real(c_double), value :: x, y, z
  end function new_four_vector_xyz
end interface
interface
  type(c_ptr) function new_four_vector_xyzt (x, y, z, t) bind(C)
    import
    real(c_double), value :: x, y, z, t
  end function new_four_vector_xyzt
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_four_vector_init
```

⟨*HepMC interface: interfaces*⟩+≡
```
interface hepmc_four_vector_init
  module procedure hepmc_four_vector_init_v4
  module procedure hepmc_four_vector_init_v3
  module procedure hepmc_four_vector_init_hepmc_prt
end interface
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_four_vector_init_v4 (pp, p)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(vector4_t), intent(in) :: p
```

```
      real(default), dimension(0:3) :: pa
      pa = vector4_get_components (p)
      pp%obj = new_four_vector_xyzt &
           (real (pa(1), c_double), &
            real (pa(2), c_double), &
            real (pa(3), c_double), &
            real (pa(0), c_double))
    end subroutine hepmc_four_vector_init_v4

    subroutine hepmc_four_vector_init_v3 (pp, p)
      type(hepmc_four_vector_t), intent(out) :: pp
      type(vector3_t), intent(in) :: p
      real(default), dimension(3) :: pa
      pa = vector3_get_components (p)
      pp%obj = new_four_vector_xyz &
           (real (pa(1), c_double), &
            real (pa(2), c_double), &
            real (pa(3), c_double))
    end subroutine hepmc_four_vector_init_v3

    subroutine hepmc_four_vector_init_hepmc_prt (pp, prt)
      type(hepmc_four_vector_t), intent(out) :: pp
      type(hepmc_particle_t), intent(in) :: prt
      pp%obj = gen_particle_momentum (prt%obj)
    end subroutine hepmc_four_vector_init_hepmc_prt
```

Here, the destructor is explicitly needed.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine four_vector_delete (p_obj) bind(C)
         import
         type(c_ptr), value :: p_obj
       end subroutine four_vector_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_four_vector_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_four_vector_final (p)
      type(hepmc_four_vector_t), intent(inout) :: p
      call four_vector_delete (p%obj)
    end subroutine hepmc_four_vector_final
```

Convert to a Lorentz vector.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function four_vector_px (p_obj) result (px) bind(C)
         import
         real(c_double) :: px
         type(c_ptr), value :: p_obj
       end function four_vector_px
    end interface
    interface
```

```
      function four_vector_py (p_obj) result (py) bind(C)
        import
        real(c_double) :: py
        type(c_ptr), value :: p_obj
      end function four_vector_py
    end interface
    interface
      function four_vector_pz (p_obj) result (pz) bind(C)
        import
        real(c_double) :: pz
        type(c_ptr), value :: p_obj
      end function four_vector_pz
    end interface
    interface
      function four_vector_e (p_obj) result (e) bind(C)
        import
        real(c_double) :: e
        type(c_ptr), value :: p_obj
      end function four_vector_e
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_four_vector_to_vector4
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_four_vector_to_vector4 (pp, p)
      type(hepmc_four_vector_t), intent(in) :: pp
      type(vector4_t), intent(out) :: p
      real(default) :: E
      real(default), dimension(3) :: p3
      E = four_vector_e (pp%obj)
      p3(1) = four_vector_px (pp%obj)
      p3(2) = four_vector_py (pp%obj)
      p3(3) = four_vector_pz (pp%obj)
      p = vector4_moving (E, vector3_moving (p3))
    end subroutine hepmc_four_vector_to_vector4
```

19.8.3 Polarization

Polarization objects are temporarily used for assigning particle polarization. We
add a flag `polarized`. If this is false, the polarization is not set and should not
be transferred to `hepmc_particle` objects.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_polarization_t
      private
      logical :: polarized = .false.
      type(c_ptr) :: obj
    end type hepmc_polarization_t
```

Constructor. The C wrapper takes polar and azimuthal angle as arguments. The Fortran version allows for either a complete polarization density matrix, or for a definite (diagonal) helicity.

HepMC does not allow to specify the degree of polarization, therefore we have to map it to either 0 or 1. We choose 0 for polarization less than 0.5 *and 1 for polarization greater than* 0.5. *Even this simplification works only for spin-1/2 and for massless particles; massive vector bosons cannot be treated this way. In particular, zero helicity is always translated as unpolarized.*

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_polarization (theta, phi) bind(C)
         import
         real(c_double), value :: theta, phi
       end function new_polarization
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_init
```

⟨*HepMC interface: interfaces*⟩+≡
```
    interface hepmc_polarization_init
       module procedure hepmc_polarization_init_pol
       module procedure hepmc_polarization_init_hel
       module procedure hepmc_polarization_init_int
    end interface
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_init_pol (hpol, pol)
      type(hepmc_polarization_t), intent(out) :: hpol
      type(polarization_t), intent(in) :: pol
      real(default) :: r, theta, phi
      if (polarization_is_polarized (pol)) then
         call polarization_to_angles (pol, r, theta, phi)
         if (r >= 0.5) then
            hpol%polarized = .true.
            hpol%obj = new_polarization &
                 (real (theta, c_double), real (phi, c_double))
         end if
      end if
    end subroutine hepmc_polarization_init_pol

    subroutine hepmc_polarization_init_hel (hpol, hel)
      type(hepmc_polarization_t), intent(out) :: hpol
      type(helicity_t), intent(in) :: hel
      integer, dimension(2) :: h
      if (helicity_is_defined (hel)) then
         h = helicity_get (hel)
         select case (h(1))
         case (1:)
            hpol%polarized = .true.
            hpol%obj = new_polarization (0._c_double, 0._c_double)
         case (:-1)
            hpol%polarized = .true.
            hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
         end select
```

2625

```
      end if
    end subroutine hepmc_polarization_init_hel

    subroutine hepmc_polarization_init_int (hpol, hel)
      type(hepmc_polarization_t), intent(out) :: hpol
      integer, intent(in) :: hel
      select case (hel)
      case (1:)
         hpol%polarized = .true.
         hpol%obj = new_polarization (0._c_double, 0._c_double)
      case (:-1)
         hpol%polarized = .true.
         hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
      end select
    end subroutine hepmc_polarization_init_int
```

Destructor. The C object is deallocated only if the `polarized` flag is set.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine polarization_delete (pol_obj) bind(C)
         import
         type(c_ptr), value :: pol_obj
       end subroutine polarization_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_final (hpol)
      type(hepmc_polarization_t), intent(inout) :: hpol
      if (hpol%polarized)  call polarization_delete (hpol%obj)
    end subroutine hepmc_polarization_final
```

Recover polarization from HepMC polarization object (with the abovementioned deficiencies).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function polarization_theta (pol_obj) result (theta) bind(C)
         import
         real(c_double) :: theta
         type(c_ptr), value :: pol_obj
       end function polarization_theta
    end interface
    interface
       function polarization_phi (pol_obj) result (phi) bind(C)
         import
         real(c_double) :: phi
         type(c_ptr), value :: pol_obj
       end function polarization_phi
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_to_pol
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_to_pol (hpol, flv, pol)
      type(hepmc_polarization_t), intent(in) :: hpol
      type(flavor_t), intent(in) :: flv
      type(polarization_t), intent(out) :: pol
      real(default) :: theta, phi
      theta = polarization_theta (hpol%obj)
      phi = polarization_phi (hpol%obj)
      call polarization_init_angles (pol, flv, 1._default, theta, phi)
    end subroutine hepmc_polarization_to_pol
```

Recover helicity. Here, ϕ is ignored and only the sign of $\cos\theta$ is relevant, mapped to positive/negative helicity.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_polarization_to_hel
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_polarization_to_hel (hpol, flv, hel)
      type(hepmc_polarization_t), intent(in) :: hpol
      type(flavor_t), intent(in) :: flv
      type(helicity_t), intent(out) :: hel
      real(default) :: theta
      integer :: hmax
      theta = polarization_theta (hpol%obj)
      hmax = flavor_get_spin_type (flv) / 2
      call helicity_init (hel, sign (hmax, nint (cos (theta))))
    end subroutine hepmc_polarization_to_hel
```

19.8.4 GenParticle

Particle objects have the obvious meaning.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_particle_t
       private
       type(c_ptr) :: obj
    end type hepmc_particle_t
```

Constructor. The C version takes a FourVector object, which in the Fortran wrapper is created on the fly from a `vector4` Lorentz vector.

No destructor is needed as long as all particles are entered into vertex containers.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function new_gen_particle (prt_obj, pdg_id, status) bind(C)
        import
        type(c_ptr), value :: prt_obj
        integer(c_int), value :: pdg_id, status
      end function new_gen_particle
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_particle_init (prt, p, pdg, status)
      type(hepmc_particle_t), intent(out) :: prt
      type(vector4_t), intent(in) :: p
      integer, intent(in) :: pdg, status
      type(hepmc_four_vector_t) :: pp
      call hepmc_four_vector_init (pp, p)
      prt%obj = new_gen_particle (pp%obj, int (pdg, c_int), int (status, c_int))
      call hepmc_four_vector_final (pp)
    end subroutine hepmc_particle_init
```

Set the particle color flow.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_particle_set_flow (prt_obj, code_index, code) bind(C)
         import
         type(c_ptr), value :: prt_obj
         integer(c_int), value :: code_index, code
       end subroutine gen_particle_set_flow
    end interface
```

Set the particle color. Either from a `color_t` object or directly from a pair of integers.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface hepmc_particle_set_color
       module procedure hepmc_particle_set_color_col
       module procedure hepmc_particle_set_color_int
    end interface hepmc_particle_set_color
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_set_color
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_particle_set_color_col (prt, col)
      type(hepmc_particle_t), intent(inout) :: prt
      type(color_t), intent(in) :: col
      integer(c_int) :: c
      c = color_get_col (col)
      if (c /= 0)  call gen_particle_set_flow (prt%obj, 1_c_int, c)
      c = color_get_acl (col)
      if (c /= 0)  call gen_particle_set_flow (prt%obj, 2_c_int, c)
    end subroutine hepmc_particle_set_color_col

    subroutine hepmc_particle_set_color_int (prt, col)
      type(hepmc_particle_t), intent(inout) :: prt
      integer, dimension(2), intent(in) :: col
      integer(c_int) :: c
      c = col(1)
      if (c /= 0)  call gen_particle_set_flow (prt%obj, 1_c_int, c)
      c = col(2)
      if (c /= 0)  call gen_particle_set_flow (prt%obj, 2_c_int, c)
    end subroutine hepmc_particle_set_color_int
```

Set the particle polarization. For the restrictions on particle polarization in HepMC, see above `hepmc_polarization_init`.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_particle_set_polarization (prt_obj, pol_obj) bind(C)
         import
         type(c_ptr), value :: prt_obj, pol_obj
       end subroutine gen_particle_set_polarization
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_set_polarization
```

⟨*HepMC interface: interfaces*⟩+≡
```
    interface hepmc_particle_set_polarization
       module procedure hepmc_particle_set_polarization_pol
       module procedure hepmc_particle_set_polarization_hel
       module procedure hepmc_particle_set_polarization_int
    end interface
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_particle_set_polarization_pol (prt, pol)
      type(hepmc_particle_t), intent(inout) :: prt
      type(polarization_t), intent(in) :: pol
      type(hepmc_polarization_t) :: hpol
      call hepmc_polarization_init (hpol, pol)
      if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
      call hepmc_polarization_final (hpol)
    end subroutine hepmc_particle_set_polarization_pol

    subroutine hepmc_particle_set_polarization_hel (prt, hel)
      type(hepmc_particle_t), intent(inout) :: prt
      type(helicity_t), intent(in) :: hel
      type(hepmc_polarization_t) :: hpol
      call hepmc_polarization_init (hpol, hel)
      if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
      call hepmc_polarization_final (hpol)
    end subroutine hepmc_particle_set_polarization_hel

    subroutine hepmc_particle_set_polarization_int (prt, hel)
      type(hepmc_particle_t), intent(inout) :: prt
      integer, intent(in) :: hel
      type(hepmc_polarization_t) :: hpol
      call hepmc_polarization_init (hpol, hel)
      if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
      call hepmc_polarization_final (hpol)
    end subroutine hepmc_particle_set_polarization_int
```

Return the HepMC barcode (unique integer ID) of the particle.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_barcode (prt_obj) result (barcode) bind(C)
         import
         integer(c_int) :: barcode
         type(c_ptr), value :: prt_obj
```

```
      end function gen_particle_barcode
   end interface
```

⟨*HepMC interface: public*⟩+≡
```
   public :: hepmc_particle_get_barcode
```

⟨*HepMC interface: procedures*⟩+≡
```
   function hepmc_particle_get_barcode (prt) result (barcode)
     integer :: barcode
     type(hepmc_particle_t), intent(in) :: prt
     barcode = gen_particle_barcode (prt%obj)
   end function hepmc_particle_get_barcode
```

Return the four-vector component of the particle object as a `vector4_t` Lorentz vector.

⟨*HepMC interface: interfaces*⟩+≡
```
   interface
      type(c_ptr) function gen_particle_momentum (prt_obj) bind(C)
        import
        type(c_ptr), value :: prt_obj
      end function gen_particle_momentum
   end interface
```

⟨*HepMC interface: public*⟩+≡
```
   public :: hepmc_particle_get_momentum
```

⟨*HepMC interface: procedures*⟩+≡
```
   function hepmc_particle_get_momentum (prt) result (p)
     type(vector4_t) :: p
     type(hepmc_particle_t), intent(in) :: prt
     type(hepmc_four_vector_t) :: pp
     call hepmc_four_vector_init (pp, prt)
     call hepmc_four_vector_to_vector4 (pp, p)
     call hepmc_four_vector_final (pp)
   end function hepmc_particle_get_momentum
```

Return the invariant mass squared of the particle object. HepMC stores the signed invariant mass (no squaring).

⟨*HepMC interface: interfaces*⟩+≡
```
   interface
      function gen_particle_generated_mass (prt_obj) result (mass) bind(C)
        import
        real(c_double) :: mass
        type(c_ptr), value :: prt_obj
      end function gen_particle_generated_mass
   end interface
```

⟨*HepMC interface: public*⟩+≡
```
   public :: hepmc_particle_get_mass_squared
```

⟨*HepMC interface: procedures*⟩+≡
```
   function hepmc_particle_get_mass_squared (prt) result (m2)
     real(default) :: m2
     type(hepmc_particle_t), intent(in) :: prt
     real(default) :: m
     m = gen_particle_generated_mass (prt%obj)
```

```
      m2 = sign (m**2, m)
    end function hepmc_particle_get_mass_squared
```

Return the PDG ID:

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_pdg_id (prt_obj) result (pdg_id) bind(C)
         import
         integer(c_int) :: pdg_id
         type(c_ptr), value :: prt_obj
       end function gen_particle_pdg_id
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_pdg
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_pdg (prt) result (pdg)
      integer :: pdg
      type(hepmc_particle_t), intent(in) :: prt
      pdg = gen_particle_pdg_id (prt%obj)
    end function hepmc_particle_get_pdg
```

Return the status code:

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_particle_status (prt_obj) result (status) bind(C)
         import
         integer(c_int) :: status
         type(c_ptr), value :: prt_obj
       end function gen_particle_status
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_status
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_status (prt) result (status)
      integer :: status
      type(hepmc_particle_t), intent(in) :: prt
      status = gen_particle_status (prt%obj)
    end function hepmc_particle_get_status
```

Return the production/decay vertex (as a pointer, no finalization necessary).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function gen_particle_production_vertex (prt_obj) bind(C)
         import
         type(c_ptr), value :: prt_obj
       end function gen_particle_production_vertex
    end interface
    interface
       type(c_ptr) function gen_particle_end_vertex (prt_obj) bind(C)
         import
```

```
        type(c_ptr), value :: prt_obj
      end function gen_particle_end_vertex
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_particle_get_production_vertex
    public :: hepmc_particle_get_decay_vertex
```

⟨HepMC interface: procedures⟩+≡
```
    function hepmc_particle_get_production_vertex (prt) result (v)
      type(hepmc_vertex_t) :: v
      type(hepmc_particle_t), intent(in) :: prt
      v%obj = gen_particle_production_vertex (prt%obj)
    end function hepmc_particle_get_production_vertex

    function hepmc_particle_get_decay_vertex (prt) result (v)
      type(hepmc_vertex_t) :: v
      type(hepmc_particle_t), intent(in) :: prt
      v%obj = gen_particle_end_vertex (prt%obj)
    end function hepmc_particle_get_decay_vertex
```

Return the number of parents/children.

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_particle_get_n_parents
    public :: hepmc_particle_get_n_children
```

⟨HepMC interface: procedures⟩+≡
```
    function hepmc_particle_get_n_parents (prt) result (n_parents)
      integer :: n_parents
      type(hepmc_particle_t), intent(in) :: prt
      type(hepmc_vertex_t) :: v
      v = hepmc_particle_get_production_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         n_parents = hepmc_vertex_get_n_in (v)
      else
         n_parents = 0
      end if
    end function hepmc_particle_get_n_parents

    function hepmc_particle_get_n_children (prt) result (n_children)
      integer :: n_children
      type(hepmc_particle_t), intent(in) :: prt
      type(hepmc_vertex_t) :: v
      v = hepmc_particle_get_decay_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         n_children = hepmc_vertex_get_n_out (v)
      else
         n_children = 0
      end if
    end function hepmc_particle_get_n_children
```

Convenience function: Return the array of parent particles for a given HepMC particle. The contents are HepMC barcodes that still have to be mapped to the particle indices.

⟨HepMC interface: public⟩+≡

```
      public :: hepmc_particle_get_parent_barcodes
      public :: hepmc_particle_get_child_barcodes
```
⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_parent_barcodes (prt) result (parent_barcode)
      type(hepmc_particle_t), intent(in) :: prt
      integer, dimension(:), allocatable :: parent_barcode
      type(hepmc_vertex_t) :: v
      type(hepmc_vertex_particle_in_iterator_t) :: it
      integer :: i
      v = hepmc_particle_get_production_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         allocate (parent_barcode (hepmc_vertex_get_n_in (v)))
         if (size (parent_barcode) /= 0) then
            call hepmc_vertex_particle_in_iterator_init (it, v)
            do i = 1, size (parent_barcode)
               parent_barcode(i) = hepmc_particle_get_barcode &
                    (hepmc_vertex_particle_in_iterator_get (it))
               call hepmc_vertex_particle_in_iterator_advance (it)
            end do
            call hepmc_vertex_particle_in_iterator_final (it)
         end if
      else
         allocate (parent_barcode (0))
      end if
    end function hepmc_particle_get_parent_barcodes

    function hepmc_particle_get_child_barcodes (prt) result (child_barcode)
      type(hepmc_particle_t), intent(in) :: prt
      integer, dimension(:), allocatable :: child_barcode
      type(hepmc_vertex_t) :: v
      type(hepmc_vertex_particle_out_iterator_t) :: it
      integer :: i
      v = hepmc_particle_get_decay_vertex (prt)
      if (hepmc_vertex_is_valid (v)) then
         allocate (child_barcode (hepmc_vertex_get_n_out (v)))
         call hepmc_vertex_particle_out_iterator_init (it, v)
         if (size (child_barcode) /= 0) then
            do i = 1, size (child_barcode)
               child_barcode(i) = hepmc_particle_get_barcode &
                    (hepmc_vertex_particle_out_iterator_get (it))
               call hepmc_vertex_particle_out_iterator_advance (it)
            end do
            call hepmc_vertex_particle_out_iterator_final (it)
         end if
      else
         allocate (child_barcode (0))
      end if
    end function hepmc_particle_get_child_barcodes
```

Return the polarization (assuming that the particle is completely polarized).
Note that the generated polarization object needs finalization.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
```

```
      type(c_ptr) function gen_particle_polarization (prt_obj) bind(C)
        import
        type(c_ptr), value :: prt_obj
      end function gen_particle_polarization
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_polarization
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_polarization (prt) result (pol)
      type(hepmc_polarization_t) :: pol
      type(hepmc_particle_t), intent(in) :: prt
      pol%obj = gen_particle_polarization (prt%obj)
    end function hepmc_particle_get_polarization
```

Return the particle color as a two-dimensional array (color, anticolor).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      function gen_particle_flow (prt_obj, code_index) result (code) bind(C)
        import
        integer(c_int) :: code
        type(c_ptr), value :: prt_obj
        integer(c_int), value :: code_index
      end function gen_particle_flow
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_particle_get_color
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_particle_get_color (prt) result (col)
      integer, dimension(2) :: col
      type(hepmc_particle_t), intent(in) :: prt
      col(1) = gen_particle_flow (prt%obj, 1)
      col(2) = - gen_particle_flow (prt%obj, 2)
    end function hepmc_particle_get_color
```

19.8.5 GenVertex

Vertices are made of particles (incoming and outgoing).

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_vertex_t
      private
      type(c_ptr) :: obj
    end type hepmc_vertex_t
```

Constructor. Two versions, one plain, one with the position in space and time (measured in mm) as argument. The Fortran version has initializer form, and the vertex position is an optional argument.

A destructor is unnecessary as long as all vertices are entered into an event container.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_gen_vertex () bind(C)
         import
       end function new_gen_vertex
    end interface
    interface
       type(c_ptr) function new_gen_vertex_pos (prt_obj) bind(C)
         import
         type(c_ptr), value :: prt_obj
       end function new_gen_vertex_pos
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_init (v, x)
      type(hepmc_vertex_t), intent(out) :: v
      type(vector4_t), intent(in), optional :: x
      type(hepmc_four_vector_t) :: pos
      if (present (x)) then
         call hepmc_four_vector_init (pos, x)
         v%obj = new_gen_vertex_pos (pos%obj)
         call hepmc_four_vector_final (pos)
      else
         v%obj = new_gen_vertex ()
      end if
    end subroutine hepmc_vertex_init
```

Return true if the vertex pointer is non-null:

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       function gen_vertex_is_valid (v_obj) result (flag) bind(C)
         import
         logical(c_bool) :: flag
         type(c_ptr), value :: v_obj
       end function gen_vertex_is_valid
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_is_valid (v) result (flag)
      logical :: flag
      type(hepmc_vertex_t), intent(in) :: v
      flag = gen_vertex_is_valid (v%obj)
    end function hepmc_vertex_is_valid
```

Add a particle to a vertex, incoming or outgoing.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
```

2635

```
    subroutine gen_vertex_add_particle_in (v_obj, prt_obj) bind(C)
      import
      type(c_ptr), value :: v_obj, prt_obj
    end subroutine gen_vertex_add_particle_in
  end interface
  interface
    subroutine gen_vertex_add_particle_out (v_obj, prt_obj) bind(C)
      import
      type(c_ptr), value :: v_obj, prt_obj
    end subroutine gen_vertex_add_particle_out
  end interface
```

⟨*HepMC interface: public*⟩+≡
```
  public :: hepmc_vertex_add_particle_in
  public :: hepmc_vertex_add_particle_out
```

⟨*HepMC interface: procedures*⟩+≡
```
  subroutine hepmc_vertex_add_particle_in (v, prt)
    type(hepmc_vertex_t), intent(inout) :: v
    type(hepmc_particle_t), intent(in) :: prt
    call gen_vertex_add_particle_in (v%obj, prt%obj)
  end subroutine hepmc_vertex_add_particle_in

  subroutine hepmc_vertex_add_particle_out (v, prt)
    type(hepmc_vertex_t), intent(inout) :: v
    type(hepmc_particle_t), intent(in) :: prt
    call gen_vertex_add_particle_out (v%obj, prt%obj)
  end subroutine hepmc_vertex_add_particle_out
```

Return the number of incoming/outgoing particles.
⟨*HepMC interface: interfaces*⟩+≡
```
  interface
    function gen_vertex_particles_in_size (v_obj) result (size) bind(C)
      import
      integer(c_int) :: size
      type(c_ptr), value :: v_obj
    end function gen_vertex_particles_in_size
  end interface
  interface
    function gen_vertex_particles_out_size (v_obj) result (size) bind(C)
      import
      integer(c_int) :: size
      type(c_ptr), value :: v_obj
    end function gen_vertex_particles_out_size
  end interface
```

⟨*HepMC interface: public*⟩+≡
```
  public :: hepmc_vertex_get_n_in
  public :: hepmc_vertex_get_n_out
```

⟨*HepMC interface: procedures*⟩+≡
```
  function hepmc_vertex_get_n_in (v) result (n_in)
    integer :: n_in
    type(hepmc_vertex_t), intent(in) :: v
    n_in = gen_vertex_particles_in_size (v%obj)
  end function hepmc_vertex_get_n_in
```

```
function hepmc_vertex_get_n_out (v) result (n_out)
  integer :: n_out
  type(hepmc_vertex_t), intent(in) :: v
  n_out = gen_vertex_particles_out_size (v%obj)
end function hepmc_vertex_get_n_out
```

19.8.6 Vertex-particle-in iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_vertex_particle_in_iterator_t
      private
      type(c_ptr) :: obj
      type(c_ptr) :: v_obj
    end type hepmc_vertex_particle_in_iterator_t
```

Constructor. The iterator is initialized at the first particle in the vertex.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function &
          new_vertex_particles_in_const_iterator (v_obj) bind(C)
        import
        type(c_ptr), value :: v_obj
      end function new_vertex_particles_in_const_iterator
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_init (it, v)
      type(hepmc_vertex_particle_in_iterator_t), intent(out) :: it
      type(hepmc_vertex_t), intent(in) :: v
      it%obj = new_vertex_particles_in_const_iterator (v%obj)
      it%v_obj = v%obj
    end subroutine hepmc_vertex_particle_in_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine vertex_particles_in_const_iterator_delete (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end subroutine vertex_particles_in_const_iterator_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_final (it)
      type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
      call vertex_particles_in_const_iterator_delete (it%obj)
    end subroutine hepmc_vertex_particle_in_iterator_final
```

Increment

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine vertex_particles_in_const_iterator_advance (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end subroutine vertex_particles_in_const_iterator_advance
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_advance (it)
      type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
      call vertex_particles_in_const_iterator_advance (it%obj)
    end subroutine hepmc_vertex_particle_in_iterator_advance
```

Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine vertex_particles_in_const_iterator_reset &
          (it_obj, v_obj) bind(C)
        import
        type(c_ptr), value :: it_obj, v_obj
      end subroutine vertex_particles_in_const_iterator_reset
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_in_iterator_reset (it)
      type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
      call vertex_particles_in_const_iterator_reset (it%obj, it%v_obj)
    end subroutine hepmc_vertex_particle_in_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      function vertex_particles_in_const_iterator_is_valid &
          (it_obj, v_obj) result (flag) bind(C)
        import
        logical(c_bool) :: flag
        type(c_ptr), value :: it_obj, v_obj
      end function vertex_particles_in_const_iterator_is_valid
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_particle_in_iterator_is_valid (it) result (flag)
      logical :: flag
      type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
      flag = vertex_particles_in_const_iterator_is_valid (it%obj, it%v_obj)
    end function hepmc_vertex_particle_in_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function &
          vertex_particles_in_const_iterator_get (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end function vertex_particles_in_const_iterator_get
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_in_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_particle_in_iterator_get (it) result (prt)
      type(hepmc_particle_t) :: prt
      type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
      prt%obj = vertex_particles_in_const_iterator_get (it%obj)
    end function hepmc_vertex_particle_in_iterator_get
```

19.8.7 Vertex-particle-out iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_vertex_particle_out_iterator_t
      private
      type(c_ptr) :: obj
      type(c_ptr) :: v_obj
    end type hepmc_vertex_particle_out_iterator_t
```

Constructor. The iterator is initialized at the first particle in the vertex.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function &
          new_vertex_particles_out_const_iterator (v_obj) bind(C)
        import
```

```
      type(c_ptr), value :: v_obj
    end function new_vertex_particles_out_const_iterator
  end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_init (it, v)
      type(hepmc_vertex_particle_out_iterator_t), intent(out) :: it
      type(hepmc_vertex_t), intent(in) :: v
      it%obj = new_vertex_particles_out_const_iterator (v%obj)
      it%v_obj = v%obj
    end subroutine hepmc_vertex_particle_out_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine vertex_particles_out_const_iterator_delete (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end subroutine vertex_particles_out_const_iterator_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_final (it)
      type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
      call vertex_particles_out_const_iterator_delete (it%obj)
    end subroutine hepmc_vertex_particle_out_iterator_final
```

Increment

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine vertex_particles_out_const_iterator_advance (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end subroutine vertex_particles_out_const_iterator_advance
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_advance (it)
      type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
      call vertex_particles_out_const_iterator_advance (it%obj)
    end subroutine hepmc_vertex_particle_out_iterator_advance
```

Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine vertex_particles_out_const_iterator_reset &
```

```
          (it_obj, v_obj) bind(C)
        import
        type(c_ptr), value :: it_obj, v_obj
      end subroutine vertex_particles_out_const_iterator_reset
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_vertex_particle_out_iterator_reset (it)
      type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
      call vertex_particles_out_const_iterator_reset (it%obj, it%v_obj)
    end subroutine hepmc_vertex_particle_out_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      function vertex_particles_out_const_iterator_is_valid &
           (it_obj, v_obj) result (flag) bind(C)
        import
        logical(c_bool) :: flag
        type(c_ptr), value :: it_obj, v_obj
      end function vertex_particles_out_const_iterator_is_valid
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_particle_out_iterator_is_valid (it) result (flag)
      logical :: flag
      type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
      flag = vertex_particles_out_const_iterator_is_valid (it%obj, it%v_obj)
    end function hepmc_vertex_particle_out_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not
be finalized, since it contains merely a pointer to the particle which is owned by
the vertex.)

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      type(c_ptr) function &
           vertex_particles_out_const_iterator_get (it_obj) bind(C)
        import
        type(c_ptr), value :: it_obj
      end function vertex_particles_out_const_iterator_get
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_vertex_particle_out_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_vertex_particle_out_iterator_get (it) result (prt)
      type(hepmc_particle_t) :: prt
      type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
      prt%obj = vertex_particles_out_const_iterator_get (it%obj)
```

```
        end function hepmc_vertex_particle_out_iterator_get
```

19.8.8   GenEvent

The main object of HepMC is a GenEvent. The object is filled by GenVertex objects, which in turn contain GenParticle objects.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_event_t
       private
       type(c_ptr) :: obj
    end type hepmc_event_t
```

Constructor. Arguments are process ID (integer) and event ID (integer).
The Fortran version has initializer form.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_gen_event (proc_id, event_id) bind(C)
         import
         integer(c_int), value :: proc_id, event_id
       end function new_gen_event
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_init (evt, proc_id, event_id)
      type(hepmc_event_t), intent(out) :: evt
      integer, intent(in), optional :: proc_id, event_id
      integer(c_int) :: pid, eid
      pid = 0;  if (present (proc_id))  pid = proc_id
      eid = 0;  if (present (event_id)) eid = event_id
      evt%obj = new_gen_event (pid, eid)
    end subroutine hepmc_event_init
```

Destructor.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_delete (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end subroutine gen_event_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_final (evt)
      type(hepmc_event_t), intent(inout) :: evt
      call gen_event_delete (evt%obj)
    end subroutine hepmc_event_final
```

Screen output. Printing to file is possible in principle (using a C++ output channel), by allowing an argument. Printing to an open Fortran unit is obviously not possible.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_print (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end subroutine gen_event_print
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_print
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_print (evt)
      type(hepmc_event_t), intent(in) :: evt
      call gen_event_print (evt%obj)
    end subroutine hepmc_event_print
```

Get the event number.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       integer(c_int) function gen_event_event_number (evt_obj) bind(C)
         use iso_c_binding !NODEP!
         type(c_ptr), value :: evt_obj
       end function gen_event_event_number
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_get_event_index
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_get_event_index (evt) result (i_proc)
      integer :: i_proc
      type(hepmc_event_t), intent(in) :: evt
      i_proc = gen_event_event_number (evt%obj)
    end function hepmc_event_get_event_index
```

Set the numeric signal process ID

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_set_signal_process_id (evt_obj, proc_id) bind(C)
         import
         type(c_ptr), value :: evt_obj
         integer(c_int), value :: proc_id
       end subroutine gen_event_set_signal_process_id
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_process_id
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_process_id (evt, proc)
      type(hepmc_event_t), intent(in) :: evt
      integer, intent(in) :: proc
      integer(c_int) :: i_proc
      i_proc = proc
      call gen_event_set_signal_process_id (evt%obj, i_proc)
    end subroutine hepmc_event_set_process_id
```

Get the numeric signal process ID

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      integer(c_int) function gen_event_signal_process_id (evt_obj) bind(C)
        import
        type(c_ptr), value :: evt_obj
      end function gen_event_signal_process_id
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_get_process_id
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_get_process_id (evt) result (i_proc)
      integer :: i_proc
      type(hepmc_event_t), intent(in) :: evt
      i_proc = gen_event_signal_process_id (evt%obj)
    end function hepmc_event_get_process_id
```

Set the event energy scale

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine gen_event_set_event_scale (evt_obj, scale) bind(C)
        import
        type(c_ptr), value :: evt_obj
        real(c_double), value :: scale
      end subroutine gen_event_set_event_scale
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_scale
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_scale (evt, scale)
      type(hepmc_event_t), intent(in) :: evt
      real(default), intent(in) :: scale
      real(c_double) :: cscale
      cscale = scale
      call gen_event_set_event_scale (evt%obj, cscale)
    end subroutine hepmc_event_set_scale
```

Get the event energy scale

⟨HepMC interface: interfaces⟩+≡
```
    interface
       real(c_double) function gen_event_event_scale (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end function gen_event_event_scale
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_event_get_scale
```

⟨HepMC interface: procedures⟩+≡
```
    function hepmc_event_get_scale (evt) result (scale)
      real(default) :: scale
      type(hepmc_event_t), intent(in) :: evt
      scale = gen_event_event_scale (evt%obj)
    end function hepmc_event_get_scale
```

Set the value of α_{QCD}.

⟨HepMC interface: interfaces⟩+≡
```
    interface
       subroutine gen_event_set_alpha_qcd (evt_obj, a) bind(C)
         import
         type(c_ptr), value :: evt_obj
         real(c_double), value :: a
       end subroutine gen_event_set_alpha_qcd
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_event_set_alpha_qcd
```

⟨HepMC interface: procedures⟩+≡
```
    subroutine hepmc_event_set_alpha_qcd (evt, alpha)
      type(hepmc_event_t), intent(in) :: evt
      real(default), intent(in) :: alpha
      real(c_double) :: a
      a = alpha
      call gen_event_set_alpha_qcd (evt%obj, a)
    end subroutine hepmc_event_set_alpha_qcd
```

Get the value of α_{QCD}.

⟨HepMC interface: interfaces⟩+≡
```
    interface
       real(c_double) function gen_event_alpha_qcd (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end function gen_event_alpha_qcd
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_event_get_alpha_qcd
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_get_alpha_qcd (evt) result (alpha)
      real(default) :: alpha
      type(hepmc_event_t), intent(in) :: evt
      alpha = gen_event_alpha_qcd (evt%obj)
    end function hepmc_event_get_alpha_qcd
```

Set the value of α_{QED}.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_set_alpha_qed (evt_obj, a) bind(C)
         import
         type(c_ptr), value :: evt_obj
         real(c_double), value :: a
       end subroutine gen_event_set_alpha_qed
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_alpha_qed
```
⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_alpha_qed (evt, alpha)
      type(hepmc_event_t), intent(in) :: evt
      real(default), intent(in) :: alpha
      real(c_double) :: a
      a = alpha
      call gen_event_set_alpha_qed (evt%obj, a)
    end subroutine hepmc_event_set_alpha_qed
```

Get the value of α_{QED}.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       real(c_double) function gen_event_alpha_qed (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end function gen_event_alpha_qed
    end interface
```
⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_get_alpha_qed
```
⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_get_alpha_qed (evt) result (alpha)
      real(default) :: alpha
      type(hepmc_event_t), intent(in) :: evt
      alpha = gen_event_alpha_qed (evt%obj)
    end function hepmc_event_get_alpha_qed
```

Clear a weight value to the end of the weight container.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_clear_weights (evt_obj) bind(C)
         use iso_c_binding !NODEP!
         type(c_ptr), value :: evt_obj
```

```
      end subroutine gen_event_clear_weights
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_clear_weights
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_clear_weights (evt)
      type(hepmc_event_t), intent(in) :: evt
      call gen_event_clear_weights (evt%obj)
    end subroutine hepmc_event_clear_weights
```

Add a weight value to the end of the weight container.
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      subroutine gen_event_add_weight (evt_obj, w) bind(C)
        use iso_c_binding !NODEP!
        type(c_ptr), value :: evt_obj
        real(c_double), value :: w
      end subroutine gen_event_add_weight
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_add_weight
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_add_weight (evt, weight)
      type(hepmc_event_t), intent(in) :: evt
      real(default), intent(in) :: weight
      real(c_double) :: w
      w = weight
      call gen_event_add_weight (evt%obj, w)
    end subroutine hepmc_event_add_weight
```

Get the size of the weight container (the number of valid elements).
⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      integer(c_int) function gen_event_weights_size (evt_obj) bind(C)
        use iso_c_binding !NODEP!
        type(c_ptr), value :: evt_obj
      end function gen_event_weights_size
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_get_weights_size
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_get_weights_size (evt) result (n)
      integer :: n
      type(hepmc_event_t), intent(in) :: evt
      n = gen_event_weights_size (evt%obj)
    end function hepmc_event_get_weights_size
```

Get the value of the weight with index `i`. (Count from 1, while C counts from zero.)

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       real(c_double) function gen_event_weight (evt_obj, i) bind(C)
         use iso_c_binding !NODEP!
         type(c_ptr), value :: evt_obj
         integer(c_int), value :: i
       end function gen_event_weight
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_get_weight
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_get_weight (evt, index) result (weight)
      real(default) :: weight
      type(hepmc_event_t), intent(in) :: evt
      integer, intent(in) :: index
      integer(c_int) :: i
      i = index - 1
      weight = gen_event_weight (evt%obj, i)
    end function hepmc_event_get_weight
```

Add a vertex to the event container.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_add_vertex (evt_obj, v_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
         type(c_ptr), value :: v_obj
       end subroutine gen_event_add_vertex
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_add_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_add_vertex (evt, v)
      type(hepmc_event_t), intent(inout) :: evt
      type(hepmc_vertex_t), intent(in) :: v
      call gen_event_add_vertex (evt%obj, v%obj)
    end subroutine hepmc_event_add_vertex
```

Mark a particular vertex as the signal process (hard interaction).

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_set_signal_process_vertex (evt_obj, v_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
         type(c_ptr), value :: v_obj
       end subroutine gen_event_set_signal_process_vertex
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_signal_process_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_signal_process_vertex (evt, v)
      type(hepmc_event_t), intent(inout) :: evt
      type(hepmc_vertex_t), intent(in) :: v
      call gen_event_set_signal_process_vertex (evt%obj, v%obj)
    end subroutine hepmc_event_set_signal_process_vertex
```

Set the beam particles explicitly.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_beam_particles
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_beam_particles (evt, prt1, prt2)
      type(hepmc_event_t), intent(inout) :: evt
      type(hepmc_particle_t), intent(in) :: prt1, prt2
      logical(c_bool) :: flag
      flag = gen_event_set_beam_particles (evt%obj, prt1%obj, prt2%obj)
    end subroutine hepmc_event_set_beam_particles
```

The C function returns a boolean which we do not use.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       logical(c_bool) function gen_event_set_beam_particles &
            (evt_obj, prt1_obj, prt2_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj, prt1_obj, prt2_obj
       end function gen_event_set_beam_particles
    end interface
```

Set the cross section and error explicitly. Note that HepMC uses pb, while WHIZARD uses fb.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_set_cross_section
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_set_cross_section (evt, xsec, xsec_err)
      type(hepmc_event_t), intent(inout) :: evt
      real(default), intent(in) :: xsec, xsec_err
      call gen_event_set_cross_section &
           (evt%obj, &
           real (xsec * 1e-3_default, c_double), &
           real (xsec_err * 1e-3_default, c_double))
    end subroutine hepmc_event_set_cross_section
```

The C function returns a boolean which we do not use.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine gen_event_set_cross_section (evt_obj, xs, xs_err) bind(C)
         import
         type(c_ptr), value :: evt_obj
         real(c_double), value :: xs, xs_err
       end subroutine gen_event_set_cross_section
```

```
    end interface
```

19.8.9 Event-particle iterator

This iterator iterates over all particles in an event. We store a pointer to the event in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_event_particle_iterator_t
       private
       type(c_ptr) :: obj
       type(c_ptr) :: evt_obj
    end type hepmc_event_particle_iterator_t
```

Constructor. The iterator is initialized at the first particle in the event.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_event_particle_const_iterator (evt_obj) bind(C)
         import
         type(c_ptr), value :: evt_obj
       end function new_event_particle_const_iterator
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_particle_iterator_init (it, evt)
      type(hepmc_event_particle_iterator_t), intent(out) :: it
      type(hepmc_event_t), intent(in) :: evt
      it%obj = new_event_particle_const_iterator (evt%obj)
      it%evt_obj = evt%obj
    end subroutine hepmc_event_particle_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       subroutine event_particle_const_iterator_delete (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end subroutine event_particle_const_iterator_delete
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_event_particle_iterator_final (it)
      type(hepmc_event_particle_iterator_t), intent(inout) :: it
      call event_particle_const_iterator_delete (it%obj)
    end subroutine hepmc_event_particle_iterator_final
```

Increment

⟨HepMC interface: interfaces⟩+≡
```
    interface
       subroutine event_particle_const_iterator_advance (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end subroutine event_particle_const_iterator_advance
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_event_particle_iterator_advance
```

⟨HepMC interface: procedures⟩+≡
```
    subroutine hepmc_event_particle_iterator_advance (it)
      type(hepmc_event_particle_iterator_t), intent(inout) :: it
      call event_particle_const_iterator_advance (it%obj)
    end subroutine hepmc_event_particle_iterator_advance
```

Reset to the beginning

⟨HepMC interface: interfaces⟩+≡
```
    interface
       subroutine event_particle_const_iterator_reset (it_obj, evt_obj) bind(C)
         import
         type(c_ptr), value :: it_obj, evt_obj
       end subroutine event_particle_const_iterator_reset
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_event_particle_iterator_reset
```

⟨HepMC interface: procedures⟩+≡
```
    subroutine hepmc_event_particle_iterator_reset (it)
      type(hepmc_event_particle_iterator_t), intent(inout) :: it
      call event_particle_const_iterator_reset (it%obj, it%evt_obj)
    end subroutine hepmc_event_particle_iterator_reset
```

Test: return true as long as we are not past the end.

⟨HepMC interface: interfaces⟩+≡
```
    interface
       function event_particle_const_iterator_is_valid &
            (it_obj, evt_obj) result (flag) bind(C)
         import
         logical(c_bool) :: flag
         type(c_ptr), value :: it_obj, evt_obj
       end function event_particle_const_iterator_is_valid
    end interface
```

⟨HepMC interface: public⟩+≡
```
    public :: hepmc_event_particle_iterator_is_valid
```

⟨HepMC interface: procedures⟩+≡
```
    function hepmc_event_particle_iterator_is_valid (it) result (flag)
      logical :: flag
      type(hepmc_event_particle_iterator_t), intent(in) :: it
      flag = event_particle_const_iterator_is_valid (it%obj, it%evt_obj)
    end function hepmc_event_particle_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function event_particle_const_iterator_get (it_obj) bind(C)
         import
         type(c_ptr), value :: it_obj
       end function event_particle_const_iterator_get
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_event_particle_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡
```
    function hepmc_event_particle_iterator_get (it) result (prt)
      type(hepmc_particle_t) :: prt
      type(hepmc_event_particle_iterator_t), intent(in) :: it
      prt%obj = event_particle_const_iterator_get (it%obj)
    end function hepmc_event_particle_iterator_get
```

19.8.10 I/O streams

There is a specific I/O stream type for handling the output of GenEvent objects (i.e., Monte Carlo event samples) to file. Opening the file is done by the constructor, closing by the destructor.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_t
```

⟨*HepMC interface: types*⟩+≡
```
    type :: hepmc_iostream_t
       private
       type(c_ptr) :: obj
    end type hepmc_iostream_t
```

Constructor for an output stream associated to a file.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
       type(c_ptr) function new_io_gen_event_out (filename) bind(C)
         import
         character(c_char), dimension(*), intent(in) :: filename
       end function new_io_gen_event_out
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_open_out
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_open_out (iostream, filename)
      type(hepmc_iostream_t), intent(out) :: iostream
      type(string_t), intent(in) :: filename
      iostream%obj = new_io_gen_event_out (char (filename) // c_null_char)
    end subroutine hepmc_iostream_open_out
```

Constructor for an input stream associated to a file.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function new_io_gen_event_in (filename) bind(C)
     import
     character(c_char), dimension(*), intent(in) :: filename
   end function new_io_gen_event_in
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_iostream_open_in
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_iostream_open_in (iostream, filename)
  type(hepmc_iostream_t), intent(out) :: iostream
  type(string_t), intent(in) :: filename
  iostream%obj = new_io_gen_event_in (char (filename) // c_null_char)
end subroutine hepmc_iostream_open_in
```

Destructor:

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine io_gen_event_delete (io_obj) bind(C)
     import
     type(c_ptr), value :: io_obj
   end subroutine io_gen_event_delete
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_iostream_close
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_iostream_close (iostream)
  type(hepmc_iostream_t), intent(inout) :: iostream
  call io_gen_event_delete (iostream%obj)
end subroutine hepmc_iostream_close
```

Write a single event to the I/O stream.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine io_gen_event_write_event (io_obj, evt_obj) bind(C)
     import
     type(c_ptr), value :: io_obj, evt_obj
   end subroutine io_gen_event_write_event
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_iostream_write_event
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_iostream_write_event (iostream, evt)
  type(hepmc_iostream_t), intent(inout) :: iostream
  type(hepmc_event_t), intent(in) :: evt
  call io_gen_event_write_event (iostream%obj, evt%obj)
end subroutine hepmc_iostream_write_event
```

Read a single event from the I/O stream. Return true if successful.

⟨*HepMC interface: interfaces*⟩+≡
```
    interface
      logical(c_bool) function io_gen_event_read_event (io_obj, evt_obj) bind(C)
        import
        type(c_ptr), value :: io_obj, evt_obj
      end function io_gen_event_read_event
    end interface
```

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_iostream_read_event
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_iostream_read_event (iostream, evt, ok)
      type(hepmc_iostream_t), intent(inout) :: iostream
      type(hepmc_event_t), intent(in) :: evt
      logical, intent(out) :: ok
      ok = io_gen_event_read_event (iostream%obj, evt%obj)
    end subroutine hepmc_iostream_read_event
```

19.8.11 Unit tests

This test example is an abridged version from the build-from-scratch example in the HepMC distribution. We create two vertices for $p \to q$ PDF splitting, then a vertex for a $qq \to W^- g$ hard-interaction process, and finally a vertex for $W^- \to qq$ decay. The setup is for LHC kinematics.

Extending the original example, we set color flow for the incoming quarks and polarization for the outgoing photon. For the latter, we have to define a particle-data object for the photon, so a flavor object can be correctly initialized.

⟨*HepMC interface: public*⟩+≡
```
    public :: hepmc_test
```

⟨*HepMC interface: procedures*⟩+≡
```
    subroutine hepmc_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*HepMC interface: execute tests*⟩
```
    end subroutine hepmc_test
```

⟨*HepMC interface: execute tests*⟩≡
```
    call test (hepmc_interface_1, "hepmc_interface_1", &
        "check HepMC interface", &
        u, results)
```

⟨*HepMC interface: tests*⟩≡
```
    subroutine hepmc_interface_1 (u)
      integer, intent(in) :: u
      integer :: u_file, iostat
      type(hepmc_event_t) :: evt
      type(hepmc_vertex_t) :: v1, v2, v3, v4
      type(hepmc_particle_t) :: prt1, prt2, prt3, prt4, prt5, prt6, prt7, prt8
      type(hepmc_iostream_t) :: iostream
      type(flavor_t) :: flv
```

```
type(color_t) :: col
type(polarization_t) :: pol
type(particle_data_t), target :: photon_data
character(80) :: buffer

write (u, "(A)")  "* Test output: HepMC interface"
write (u, "(A)")  "*   Purpose: test HepMC interface"
write (u, "(A)")

write (u, "(A)")  "* Initialization"
write (u, "(A)")

! Initialize a photon flavor object and some polarization
call particle_data_init (photon_data, var_str ("PHOTON"), 22)
call particle_data_set (photon_data, spin_type=VECTOR)
call particle_data_freeze (photon_data)
call flavor_init (flv, photon_data)
call polarization_init_angles &
     (pol, flv, 0.6_default, 1._default, 0.5_default)

! Event initialization
call hepmc_event_init (evt, 20, 1)

write (u, "(A)")  "* p -> q splitting"
write (u, "(A)")

! $p\to q$ splittings
call hepmc_vertex_init (v1)
call hepmc_event_add_vertex (evt, v1)
call hepmc_vertex_init (v2)
call hepmc_event_add_vertex (evt, v2)
call particle_init (prt1, &
    0._default, 0._default, 7000._default, 7000._default, &
    2212, 3)
call hepmc_vertex_add_particle_in (v1, prt1)
call particle_init (prt2, &
    0._default, 0._default,-7000._default, 7000._default, &
    2212, 3)
call hepmc_vertex_add_particle_in (v2, prt2)
call particle_init (prt3, &
    .750_default, -1.569_default, 32.191_default, 32.238_default, &
    1, 3)
call color_init_from_array (col, [501])
call hepmc_particle_set_color (prt3, col)
call hepmc_vertex_add_particle_out (v1, prt3)
call particle_init (prt4, &
    -3.047_default, -19._default, -54.629_default, 57.920_default, &
    -2, 3)
call color_init_from_array (col, [-501])
call hepmc_particle_set_color (prt4, col)
call hepmc_vertex_add_particle_out (v2, prt4)

write (u, "(A)")  "* Hard interaction"
write (u, "(A)")
```

```
! Hard interaction
call hepmc_vertex_init (v3)
call hepmc_event_add_vertex (evt, v3)
call hepmc_vertex_add_particle_in (v3, prt3)
call hepmc_vertex_add_particle_in (v3, prt4)
call particle_init (prt6, &
     -3.813_default, 0.113_default, -1.833_default, 4.233_default, &
     22, 1)
call hepmc_particle_set_polarization (prt6, pol)
call hepmc_vertex_add_particle_out (v3, prt6)
call particle_init (prt5, &
     1.517_default, -20.68_default, -20.605_default, 85.925_default, &
     -24, 3)
call hepmc_vertex_add_particle_out (v3, prt5)
call hepmc_event_set_signal_process_vertex (evt, v3)

! $W^-$ decay
call vertex_init_pos (v4, &
     0.12_default, -0.3_default, 0.05_default, 0.004_default)
call hepmc_event_add_vertex (evt, v4)
call hepmc_vertex_add_particle_in (v4, prt5)
call particle_init (prt7, &
     -2.445_default, 28.816_default, 6.082_default, 29.552_default, &
     1, 1)
call hepmc_vertex_add_particle_out (v4, prt7)
call particle_init (prt8, &
     3.962_default, -49.498_default, -26.687_default, 56.373_default, &
     -2, 1)
call hepmc_vertex_add_particle_out (v4, prt8)

! Event output
call hepmc_event_print (evt)
write (u, "(A)")  "Writing to file 'hepmc_test.hepmc'"
write (u, "(A)")

call hepmc_iostream_open_out (iostream , var_str ("hepmc_test.hepmc"))
call hepmc_iostream_write_event (iostream, evt)
call hepmc_iostream_close (iostream)

write (u, "(A)")  "Writing completed"

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = "hepmc_test.hepmc", &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:14) == "HepMC::Version")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
```

```
      end do
      close (u_file)

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"
      write (u, "(A)")

      ! Wrapup
      call polarization_final (pol)
      call hepmc_event_final (evt)

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: hepmc_interface_1"

   contains

     subroutine vertex_init_pos (v, x, y, z, t)
       type(hepmc_vertex_t), intent(out) :: v
       real(default), intent(in) :: x, y, z, t
       type(vector4_t) :: xx
       xx = vector4_moving (t, vector3_moving ([x, y, z]))
       call hepmc_vertex_init (v, xx)
     end subroutine vertex_init_pos

     subroutine particle_init (prt, px, py, pz, E, pdg, status)
       type(hepmc_particle_t), intent(out) :: prt
       real(default), intent(in) :: px, py, pz, E
       integer, intent(in) :: pdg, status
       type(vector4_t) :: p
       p = vector4_moving (E, vector3_moving ([px, py, pz]))
       call hepmc_particle_init (prt, p, pdg, status)
     end subroutine particle_init

   end subroutine hepmc_interface_1
```

19.9   HepMC Output

The HepMC event record is standardized. It is an ASCII format. We try our
best at using it for both input and output.

⟨eio_hepmc.f90⟩≡
  ⟨*File header*⟩

```
  module eio_hepmc

    use kinds !NODEP!
```
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
```
    use diagnostics !NODEP!
    use unit_tests

    use lorentz !NODEP!
```

```
    use models
    use particles
    use subevents
    use beams
    use processes
    use events
    use eio_data
    use eio_base
    use hepmc_interface
```

⟨*Standard module head*⟩

⟨*EIO HepMC: public*⟩

⟨*EIO HepMC: types*⟩

```
contains
```

⟨*EIO HepMC: procedures*⟩

⟨*EIO HepMC: tests*⟩

```
end module eio_hepmc
```

19.9.1   Type

⟨*EIO HepMC: public*⟩≡
```
    public :: eio_hepmc_t
```

⟨*EIO HepMC: types*⟩≡
```
    type, extends (eio_t) :: eio_hepmc_t
      logical :: writing = .false.
      logical :: keep_beams = .false.
      type(hepmc_iostream_t) :: iostream
     contains
     ⟨EIO HepMC: eio hepmc: TBP⟩
    end type eio_hepmc_t
```

19.9.2   Specific Methods

Set parameters that are specifically used with HepMC.

⟨*EIO HepMC: eio hepmc: TBP*⟩≡
```
    procedure :: set_parameters => eio_hepmc_set_parameters
```

⟨*EIO HepMC: procedures*⟩≡
```
    subroutine eio_hepmc_set_parameters (eio, keep_beams, extension)
      class(eio_hepmc_t), intent(inout) :: eio
      logical, intent(in), optional :: keep_beams
      type(string_t), intent(in), optional :: extension
      if (present (keep_beams))  eio%keep_beams = keep_beams
      if (present (extension)) then
         eio%extension = extension
      else
```

```
      eio%extension = "hepmc"
   end if
end subroutine eio_hepmc_set_parameters
```

19.9.3   Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡
```
   procedure :: write => eio_hepmc_write
```

⟨*EIO HepMC: procedures*⟩+≡
```
   subroutine eio_hepmc_write (object, unit)
     class(eio_hepmc_t), intent(in) :: object
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit)
     write (u, "(1x,A)")  "HepMC event stream:"
     if (object%writing) then
        write (u, "(3x,A,A)")  "Writing to file  = ", char (object%filename)
     else
        write (u, "(3x,A)")  "[closed]"
     end if
     write (u, "(3x,A,L1)")    "Keep beams        = ", object%keep_beams
   end subroutine eio_hepmc_write
```

Finalizer: close any open file.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡
```
   procedure :: final => eio_hepmc_final
```

⟨*EIO HepMC: procedures*⟩+≡
```
   subroutine eio_hepmc_final (object)
     class(eio_hepmc_t), intent(inout) :: object
     if (object%writing) then
        write (msg_buffer, "(A,A,A)")  "Events: closing HepMC file '", &
             char (object%filename), "'"
        call msg_message ()
        call hepmc_iostream_close (object%iostream)
        object%writing = .false.
     end if
   end subroutine eio_hepmc_final
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡
```
   procedure :: split_out => eio_hepmc_split_out
```

⟨*EIO HepMC: procedures*⟩+≡
```
   subroutine eio_hepmc_split_out (eio)
     class(eio_hepmc_t), intent(inout) :: eio
     if (eio%split) then
        eio%split_index = eio%split_index + 1
```

```
      call eio%set_filename ()
      write (msg_buffer, "(A,A,A)")  "Events: writing to HepMC file '", &
           char (eio%filename), "'"
      call msg_message ()
      call hepmc_iostream_close (eio%iostream)
      call hepmc_iostream_open_out (eio%iostream, eio%filename)
    end if
  end subroutine eio_hepmc_split_out
```

Initialize event writing.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡
```
    procedure :: init_out => eio_hepmc_init_out
```

⟨*EIO HepMC: procedures*⟩+≡
```
  subroutine eio_hepmc_init_out (eio, sample, process_ptr, data, success)
    class(eio_hepmc_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    if (.not. present (data)) &
         call msg_bug ("HepMC initialization: missing data")
    if (data%n_beam /= 2) &
         call msg_fatal ("HepMC: defined for scattering processes only")
    if (.not. data%unweighted) &
         call msg_fatal ("HepMC: events must be unweighted")
    call eio%set_splitting (data)
    eio%sample = sample
    eio%extension = "hepmc"
    call eio%set_filename ()
    write (msg_buffer, "(A,A,A)")  "Events: writing to HepMC file '", &
         char (eio%filename), "'"
    call msg_message ()
    eio%writing = .true.
    call hepmc_iostream_open_out (eio%iostream, eio%filename)
    if (present (success))  success = .true.
  end subroutine eio_hepmc_init_out
```

Initialize event reading.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡
```
    procedure :: init_in => eio_hepmc_init_in
```

⟨*EIO HepMC: procedures*⟩+≡
```
  subroutine eio_hepmc_init_in &
       (eio, sample, process_ptr, data, success, extension)
    class(eio_hepmc_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(process_ptr_t), dimension(:), intent(in) :: process_ptr
    type(event_sample_data_t), intent(inout), optional :: data
    logical, intent(out), optional :: success
    call msg_bug ("HepMC: event input not supported")
    if (present (success))  success = .false.
  end subroutine eio_hepmc_init_in
```

Switch from input to output: reopen the file for reading.

⟨EIO HepMC: eio hepmc: TBP⟩+≡
```
procedure :: switch_inout => eio_hepmc_switch_inout
```

⟨EIO HepMC: procedures⟩+≡
```
subroutine eio_hepmc_switch_inout (eio, success)
  class(eio_hepmc_t), intent(inout) :: eio
  logical, intent(out), optional :: success
  call msg_bug ("HepMC: in-out switch not supported")
  if (present (success))  success = .false.
end subroutine eio_hepmc_switch_inout
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

⟨EIO HepMC: eio hepmc: TBP⟩+≡
```
procedure :: output => eio_hepmc_output
```

⟨EIO HepMC: procedures⟩+≡
```
subroutine eio_hepmc_output (eio, event, i_prc, reading)
  class(eio_hepmc_t), intent(inout) :: eio
  type(event_t), intent(in), target :: event
  integer, intent(in) :: i_prc
  logical, intent(in), optional :: reading
  type(hepmc_event_t) :: hepmc_event
  type(particle_set_t), pointer :: pset_ptr
  if (eio%writing) then
     pset_ptr => event%get_particle_set_ptr ()
     call hepmc_event_init (hepmc_event, proc_id = i_prc)
     call hepmc_event_from_particle_set (hepmc_event, pset_ptr)
     call hepmc_iostream_write_event (eio%iostream, hepmc_event)
     call hepmc_event_final (hepmc_event)
  else
     call eio%write ()
     call msg_fatal ("HepMC file is not open for writing")
  end if
end subroutine eio_hepmc_output
```

Input an event.

⟨EIO HepMC: eio hepmc: TBP⟩+≡
```
procedure :: input_i_prc => eio_hepmc_input_i_prc
procedure :: input_event => eio_hepmc_input_event
```

⟨EIO HepMC: procedures⟩+≡
```
subroutine eio_hepmc_input_i_prc (eio, i_prc, iostat)
  class(eio_hepmc_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  call msg_bug ("HepMC: event input not supported")
  i_prc = 0
  iostat = 1
end subroutine eio_hepmc_input_i_prc
```

```
subroutine eio_hepmc_input_event (eio, event, iostat)
  class(eio_hepmc_t), intent(inout) :: eio
  type(event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  call msg_bug ("HepMC: event input not supported")
  iostat = 1
end subroutine eio_hepmc_input_event
```

19.9.4   Unit tests

⟨*EIO HepMC: public*⟩+≡
```
public :: eio_hepmc_test
```

⟨*EIO HepMC: tests*⟩≡
```
subroutine eio_hepmc_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*EIO HepMC: execute tests*⟩
```
end subroutine eio_hepmc_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO HepMC: execute tests*⟩≡
```
call test (eio_hepmc_1, "eio_hepmc_1", &
     "read and write event contents", &
     u, results)
```

⟨*EIO HepMC: tests*⟩+≡
```
subroutine eio_hepmc_1 (u)
  integer, intent(in) :: u
  type(model_list_t) :: model_list
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_ptr_t) :: process_ptr
  type(process_instance_t), allocatable, target :: process_instance
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat
  character(116) :: buffer

  write (u, "(A)")  "* Test output: eio_hepmc_1"
  write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
  write (u, "(A)")

  call syntax_model_file_init ()

  write (u, "(A)")  "* Initialize test process"

  allocate (process)
  process_ptr%ptr => process
```

```
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_hepmc_1"

allocate (eio_hepmc_t :: eio)

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents (blanking out last two digits):"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".hepmc"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   if (trim (buffer) == "")  cycle
   if (buffer(1:14) == "HepMC::Version")  cycle
   if (buffer(1:10) == "P 10001 25") &
       call buffer_blanker (buffer, 32, 55, 78)
   if (buffer(1:10) == "P 10002 25") &
       call buffer_blanker (buffer, 33, 56, 79)
   if (buffer(1:10) == "P 10003 25") &
       call buffer_blanker (buffer, 29, 53, 78, 101)
   if (buffer(1:10) == "P 10004 25") &
       call buffer_blanker (buffer, 28, 51, 76, 99)
   write (u, "(A)") trim (buffer)
end do
close (u_file)
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Reset data"
  write (u, "(A)")

  deallocate (eio)
  allocate (eio_hepmc_t :: eio)

  select type (eio)
  type is (eio_hepmc_t)
     call eio%set_parameters (keep_beams = .true.)
  end select
  call eio%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call event%final ()
  deallocate (event)

  call cleanup_test_process (process, process_instance)
  deallocate (process_instance)
  deallocate (process)

  call model_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: eio_hepmc_1"

contains

  subroutine buffer_blanker (buf, pos1, pos2, pos3, pos4)
    character(len=*), intent(inout) :: buf
    integer, intent(in) :: pos1, pos2, pos3
    integer, intent(in), optional :: pos4
    type(string_t) :: line
    line = var_str (trim (buf))
    line = replace (line, pos1, "XX")
    line = replace (line, pos2, "XX")
    line = replace (line, pos3, "XX")
    if (present (pos4)) then
       line = replace (line, pos4, "XX")
    end if
    line = replace (line, "4999999999999", "5000000000000")
    buf = char (line)
  end subroutine buffer_blanker

end subroutine eio_hepmc_1
```

19.10 ASCII File Formats

Here, we implement several ASCII file formats. It is possible to switch between them using flags.

⟨eio_ascii.f90⟩≡
  ⟨*File header*⟩

  module eio_ascii

    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use diagnostics !NODEP!
    use unit_tests

    use lorentz !NODEP!
    use models
    use particles
    use beams
    use processes
    use events
    use eio_data
    use eio_base
    use hep_common
    use hep_events

  ⟨*Standard module head*⟩

  ⟨*EIO ascii: public*⟩

  ⟨*EIO ascii: types*⟩

    contains

  ⟨*EIO ascii: procedures*⟩

  ⟨*EIO ascii: tests*⟩

    end module eio_ascii

19.10.1 Type

⟨*EIO ascii: public*⟩≡
    public :: eio_ascii_t

⟨*EIO ascii: types*⟩≡
    type, abstract, extends (eio_t) :: eio_ascii_t
      logical :: writing = .false.
      integer :: unit = 0
      logical :: keep_beams = .false.
    contains
    ⟨*EIO ascii: eio ascii: TBP*⟩
    end type eio_ascii_t

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_ascii_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_ascii_t
    end type eio_ascii_ascii_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_athena_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_athena_t
    end type eio_ascii_athena_t
```

The debug format has a few options that can be controlled by Sindarin variables.
⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_debug_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_debug_t
       logical :: show_process = .true.
       logical :: show_transforms = .true.
       logical :: show_decay = .true.
       logical :: verbose = .true.
    end type eio_ascii_debug_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_hepevt_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_hepevt_t
    end type eio_ascii_hepevt_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_hepevt_verb_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_hepevt_verb_t
    end type eio_ascii_hepevt_verb_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_lha_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_lha_t
    end type eio_ascii_lha_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_lha_verb_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_lha_verb_t
    end type eio_ascii_lha_verb_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_long_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_long_t
    end type eio_ascii_long_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_mokka_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_mokka_t
    end type eio_ascii_mokka_t
```

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_short_t
```

⟨*EIO ascii: types*⟩+≡
```
    type, extends (eio_ascii_t) :: eio_ascii_short_t
    end type eio_ascii_short_t
```

19.10.2 Specific Methods

Set parameters that are specifically used with ASCII file formats. In particular,
this is the file extension.

⟨*EIO ascii: eio ascii: TBP*⟩≡
```
    procedure :: set_parameters => eio_ascii_set_parameters
```

⟨*EIO ascii: procedures*⟩≡
```
    subroutine eio_ascii_set_parameters (eio, keep_beams, extension, &
        show_process, show_transforms, show_decay, verbose)
      class(eio_ascii_t), intent(inout) :: eio
      logical, intent(in), optional :: keep_beams
      type(string_t), intent(in), optional :: extension
      logical, intent(in), optional :: show_process, show_transforms, show_decay
      logical, intent(in), optional :: verbose
      if (present (keep_beams))  eio%keep_beams = keep_beams
      if (present (extension)) then
         eio%extension = extension
      else
         select type (eio)
         type is (eio_ascii_ascii_t)
            eio%extension = "evt"
         type is (eio_ascii_athena_t)
            eio%extension = "athena.evt"
         type is (eio_ascii_debug_t)
            eio%extension = "debug"
         type is (eio_ascii_hepevt_t)
            eio%extension = "hepevt"
         type is (eio_ascii_hepevt_verb_t)
            eio%extension = "hepevt.verb"
         type is (eio_ascii_lha_t)
            eio%extension = "lha"
```

```
      type is (eio_ascii_lha_verb_t)
         eio%extension = "lha.verb"
      type is (eio_ascii_long_t)
         eio%extension = "long.evt"
      type is (eio_ascii_mokka_t)
         eio%extension = "mokka.evt"
      type is (eio_ascii_short_t)
         eio%extension = "short.evt"
      end select
   end if
   select type (eio)
   type is (eio_ascii_debug_t)
      if (present (show_process))  eio%show_process = show_process
      if (present (show_transforms))  eio%show_transforms = show_transforms
      if (present (show_decay))  eio%show_decay = show_decay
      if (present (verbose))  eio%verbose = verbose
   end select
end subroutine eio_ascii_set_parameters
```

19.10.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
   procedure :: write => eio_ascii_write
```

⟨*EIO ascii: procedures*⟩+≡
```
   subroutine eio_ascii_write (object, unit)
     class(eio_ascii_t), intent(in) :: object
     integer, intent(in), optional :: unit
     integer :: u
     u = output_unit (unit)
     select type (object)
     type is (eio_ascii_ascii_t)
        write (u, "(1x,A)")  "ASCII event stream (default format):"
     type is (eio_ascii_athena_t)
        write (u, "(1x,A)")  "ASCII event stream (ATHENA format):"
     type is (eio_ascii_debug_t)
        write (u, "(1x,A)")  "ASCII event stream (Debugging format):"
     type is (eio_ascii_hepevt_t)
        write (u, "(1x,A)")  "ASCII event stream (HEPEVT format):"
     type is (eio_ascii_hepevt_verb_t)
        write (u, "(1x,A)")  "ASCII event stream (verbose HEPEVT format):"
     type is (eio_ascii_lha_t)
        write (u, "(1x,A)")  "ASCII event stream (LHA format):"
     type is (eio_ascii_lha_verb_t)
        write (u, "(1x,A)")  "ASCII event stream (verbose LHA format):"
     type is (eio_ascii_long_t)
        write (u, "(1x,A)")  "ASCII event stream (long format):"
     type is (eio_ascii_mokka_t)
        write (u, "(1x,A)")  "ASCII event stream (MOKKA format):"
     type is (eio_ascii_short_t)
        write (u, "(1x,A)")  "ASCII event stream (short format):"
```

```
      end select
      if (object%writing) then
         write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
      else
         write (u, "(3x,A)")  "[closed]"
      end if
      write (u, "(3x,A,L1)")    "Keep beams        = ", object%keep_beams
      select type (object)
      type is (eio_ascii_debug_t)
         write (u, "(3x,A,L1)")    "Show process      = ", object%show_process
         write (u, "(3x,A,L1)")    "Show transforms   = ", object%show_transforms
         write (u, "(3x,A,L1)")    "Show decay tree   = ", object%show_decay
         write (u, "(3x,A,L1)")    "Verbose output    = ", object%verbose
      end select
    end subroutine eio_ascii_write
```

Finalizer: close any open file.

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
    procedure :: final => eio_ascii_final
```

⟨*EIO ascii: procedures*⟩+≡
```
    subroutine eio_ascii_final (object)
      class(eio_ascii_t), intent(inout) :: object
      if (object%writing) then
         write (msg_buffer, "(A,A,A)")  "Events: closing ASCII file '", &
              char (object%filename), "'"
         call msg_message ()
         close (object%unit)
         object%writing = .false.
      end if
    end subroutine eio_ascii_final
```

Initialize event writing.

Check weight normalization. This applies to all ASCII-type files that use the HEPRUP common block. We can't allow normalization conventions that are not covered by the HEPRUP definition.

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
    procedure :: init_out => eio_ascii_init_out
```

⟨*EIO ascii: procedures*⟩+≡
```
    subroutine eio_ascii_init_out (eio, sample, process_ptr, data, success)
      class(eio_ascii_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(in), optional :: data
      logical, intent(out), optional :: success
      integer :: i
      if (.not. present (data)) &
           call msg_bug ("ASCII initialization: missing data")
      if (data%n_beam /= 2) &
           call msg_fatal ("ASCII: defined for scattering processes only")
      eio%sample = sample
      call eio%check_normalization (data)
      call eio%set_splitting (data)
```

```
       call eio%set_filename ()
       eio%unit = free_unit ()
       write (msg_buffer, "(A,A,A)")  "Events: writing to ASCII file '", &
            char (eio%filename), "'"
       call msg_message ()
       eio%writing = .true.
       open (eio%unit, file = char (eio%filename), &
            action = "write", status = "replace")
       select type (eio)
       type is (eio_ascii_lha_t)
          call heprup_init &
               (data%pdg_beam, &
               data%energy_beam, &
               n_processes = data%n_proc, &
               unweighted = data%unweighted, &
               negative_weights = data%negative_weights)
          do i = 1, data%n_proc
             call heprup_set_process_parameters (i = i, &
                  process_id = data%proc_num_id(i), &
                  cross_section = data%cross_section(i), &
                  error = data%error(i))
          end do
          call heprup_write_ascii (eio%unit)
       type is (eio_ascii_lha_verb_t)
          call heprup_init &
               (data%pdg_beam, &
               data%energy_beam, &
               n_processes = data%n_proc, &
               unweighted = data%unweighted, &
               negative_weights = data%negative_weights)
          do i = 1, data%n_proc
             call heprup_set_process_parameters (i = i, &
                  process_id = data%proc_num_id(i), &
                  cross_section = data%cross_section(i), &
                  error = data%error(i))
          end do
          call heprup_write_verbose (eio%unit)
       end select
       if (present (success))  success = .true.
     end subroutine eio_ascii_init_out
```

Some event properties do not go well with some output formats. In particular, many formats require unweighted events.

⟨EIO ascii: eio ascii: TBP⟩+≡
```
    procedure :: check_normalization => eio_ascii_check_normalization
```

⟨EIO ascii: procedures⟩+≡
```
    subroutine eio_ascii_check_normalization (eio, data)
      class(eio_ascii_t), intent(in) :: eio
      type(event_sample_data_t), intent(in) :: data
      if (data%unweighted) then
      else
         select type (eio)
         type is (eio_ascii_athena_t);  call msg_fatal &
```

```
                  ("Event output (Athena format): events must be unweighted.")
           type is (eio_ascii_hepevt_t);  call msg_fatal &
                  ("Event output (HEPEVT format): events must be unweighted.")
           type is (eio_ascii_hepevt_verb_t);  call msg_fatal &
                  ("Event output (HEPEVT format): events must be unweighted.")
           end select
           select case (data%norm_mode)
           case (NORM_SIGMA)
           case default
              select type (eio)
              type is (eio_ascii_lha_t)
                 call msg_fatal &
                      ("Event output (LHA): normalization for weighted events &
                      &must be 'sigma'")
              type is (eio_ascii_lha_verb_t)
                 call msg_fatal &
                      ("Event output (LHA): normalization for weighted events &
                      &must be 'sigma'")
              end select
           end select
       end if
    end subroutine eio_ascii_check_normalization
```

Initialize event reading.

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
    procedure :: init_in => eio_ascii_init_in
```

⟨*EIO ascii: procedures*⟩+≡
```
    subroutine eio_ascii_init_in &
        (eio, sample, process_ptr, data, success, extension)
      class(eio_ascii_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      call msg_bug ("ASCII: event input not supported")
      if (present (success))  success = .false.
    end subroutine eio_ascii_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
    procedure :: switch_inout => eio_ascii_switch_inout
```

⟨*EIO ascii: procedures*⟩+≡
```
    subroutine eio_ascii_switch_inout (eio, success)
      class(eio_ascii_t), intent(inout) :: eio
      logical, intent(out), optional :: success
      call msg_bug ("ASCII: in-out switch not supported")
      if (present (success))  success = .false.
    end subroutine eio_ascii_switch_inout
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file. (We assume that the common block contents are still intact.)

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
    procedure :: split_out => eio_ascii_split_out
```

⟨*EIO ascii: procedures*⟩+≡
```
    subroutine eio_ascii_split_out (eio)
      class(eio_ascii_t), intent(inout) :: eio
      if (eio%split) then
         eio%split_index = eio%split_index + 1
         call eio%set_filename ()
         write (msg_buffer, "(A,A,A)")  "Events: writing to ASCII file '", &
             char (eio%filename), "'"
         call msg_message ()
         close (eio%unit)
         open (eio%unit, file = char (eio%filename), &
             action = "write", status = "replace")
         select type (eio)
         type is (eio_ascii_lha_t)
            call heprup_write_ascii (eio%unit)
         type is (eio_ascii_lha_verb_t)
            call heprup_write_verbose (eio%unit)
         end select
      end if
    end subroutine eio_ascii_split_out
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

⟨*EIO ascii: eio ascii: TBP*⟩+≡
```
    procedure :: output => eio_ascii_output
```

⟨*EIO ascii: procedures*⟩+≡
```
    subroutine eio_ascii_output (eio, event, i_prc, reading)
      class(eio_ascii_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      integer, intent(in) :: i_prc
      logical, intent(in), optional :: reading
      if (eio%writing) then
         select type (eio)
         type is (eio_ascii_lha_t)
            call hepeup_from_event (event, &
                process_index = i_prc, &
                keep_beams = eio%keep_beams)
            call hepeup_write_lha (eio%unit)
         type is (eio_ascii_lha_verb_t)
            call hepeup_from_event (event, &
                process_index = i_prc, &
                keep_beams = eio%keep_beams)
            call hepeup_write_verbose (eio%unit)
         type is (eio_ascii_ascii_t)
            call event%write (eio%unit, &
                show_process = .false., &
                show_transforms = .false., &
```

```
                    show_decay = .false., &
                    verbose = .false.)
         type is (eio_ascii_athena_t)
            call hepevt_from_event (event, &
                 i_evt = event%expr%index, &
                 keep_beams = eio%keep_beams)
            call hepevt_write_athena (eio%unit)
         type is (eio_ascii_debug_t)
            call event%write (eio%unit, &
                 show_process = eio%show_process, &
                 show_transforms = eio%show_transforms, &
                 show_decay = eio%show_decay, &
                 verbose = eio%verbose)
         type is (eio_ascii_hepevt_t)
            call hepevt_from_event (event, &
                 i_evt = event%expr%index, &
                 keep_beams = eio%keep_beams)
            call hepevt_write_hepevt (eio%unit)
         type is (eio_ascii_hepevt_verb_t)
            call hepevt_from_event (event, &
                 i_evt = event%expr%index, &
                 keep_beams = eio%keep_beams)
            call hepevt_write_verbose (eio%unit)
         type is (eio_ascii_long_t)
            call hepevt_from_event (event, &
                 i_evt = event%expr%index, &
                 keep_beams = eio%keep_beams)
            call hepevt_write_ascii (eio%unit, .true.)
         type is (eio_ascii_mokka_t)
            call hepevt_from_event (event, &
                 i_evt = event%expr%index, &
                 keep_beams = eio%keep_beams)
            call hepevt_write_mokka (eio%unit)
         type is (eio_ascii_short_t)
            call hepevt_from_event (event, &
                 i_evt = event%expr%index, &
                 keep_beams = eio%keep_beams)
            call hepevt_write_ascii (eio%unit, .false.)
         end select
      else
         call eio%write ()
         call msg_fatal ("ASCII file is not open for writing")
      end if
    end subroutine eio_ascii_output
```

Input an event.

⟨EIO ascii: eio ascii: TBP⟩+≡
```
    procedure :: input_i_prc => eio_ascii_input_i_prc
    procedure :: input_event => eio_ascii_input_event
```

⟨EIO ascii: procedures⟩+≡
```
    subroutine eio_ascii_input_i_prc (eio, i_prc, iostat)
      class(eio_ascii_t), intent(inout) :: eio
      integer, intent(out) :: i_prc
```

```
    integer, intent(out) :: iostat
    call msg_bug ("ASCII: event input not supported")
    i_prc = 0
    iostat = 1
  end subroutine eio_ascii_input_i_prc

  subroutine eio_ascii_input_event (eio, event, iostat)
    class(eio_ascii_t), intent(inout) :: eio
    type(event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
    call msg_bug ("ASCII: event input not supported")
    iostat = 1
  end subroutine eio_ascii_input_event
```

19.10.4   Unit tests

⟨*EIO ascii: public*⟩+≡
```
    public :: eio_ascii_test
```

⟨*EIO ascii: tests*⟩≡
```
    subroutine eio_ascii_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*EIO ascii: execute tests*⟩
```
    end subroutine eio_ascii_test
```

Test I/O methods

We test the implementation of all I/O methods, method `ascii`:

⟨*EIO ascii: execute tests*⟩≡
```
    call test (eio_ascii_1, "eio_ascii_1", &
         "read and write event contents, format [ascii]", &
         u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
    subroutine eio_ascii_1 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_1"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII ascii format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")
```

```
call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_1"

allocate (eio_ascii_ascii_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
```

```
      if (iostat /= 0)  exit
      write (u, "(A)") trim (buffer)
   end do
   close (u_file)

   write (u, "(A)")
   write (u, "(A)")  "* Reset data"
   write (u, "(A)")

   deallocate (eio)
   allocate (eio_ascii_ascii_t :: eio)

   select type (eio)
   type is (eio_ascii_ascii_t)
      call eio%set_parameters (keep_beams = .true.)
   end select
   call eio%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call event%final ()
   deallocate (event)

   call cleanup_test_process (process, process_instance)
   deallocate (process_instance)
   deallocate (process)

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: eio_ascii_1"

end subroutine eio_ascii_1
```

We test the implementation of all I/O methods, method `athena`:

⟨*EIO ascii: execute tests*⟩+≡
```
   call test (eio_ascii_2, "eio_ascii_2", &
      "read and write event contents, format [athena]", &
      u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
   subroutine eio_ascii_2 (u)
     integer, intent(in) :: u
     type(model_list_t) :: model_list
     type(event_t), allocatable, target :: event
     type(process_t), allocatable, target :: process
     type(process_ptr_t) :: process_ptr
     type(process_instance_t), allocatable, target :: process_instance
     type(event_sample_data_t) :: data
     class(eio_t), allocatable :: eio
     type(string_t) :: sample
     integer :: u_file, iostat
```

```
character(80) :: buffer

write (u, "(A)")  "* Test output: eio_ascii_2"
write (u, "(A)")  "*   Purpose: generate an event in ASCII athena format"
write (u, "(A)")  "*      and write weight to file"
write (u, "(A)")

call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_2"

allocate (eio_ascii_athena_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")
```

```
      u_file = free_unit ()
      open (u_file, file = char(sample // ".athena.evt"), &
           action = "read", status = "old")
      do
         read (u_file, "(A)", iostat = iostat)  buffer
         if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
         if (iostat /= 0)  exit
         write (u, "(A)") trim (buffer)
      end do
      close (u_file)

      write (u, "(A)")
      write (u, "(A)")  "* Reset data"
      write (u, "(A)")

      deallocate (eio)
      allocate (eio_ascii_athena_t :: eio)

      select type (eio)
      type is (eio_ascii_athena_t)
         call eio%set_parameters (keep_beams = .true.)
      end select
      call eio%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_ascii_2"

    end subroutine eio_ascii_2
```

We test the implementation of all I/O methods, method `debug`:

⟨*EIO ascii: execute tests*⟩+≡
```
    call test (eio_ascii_3, "eio_ascii_3", &
         "read and write event contents, format [debug]", &
         u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
    subroutine eio_ascii_3 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
```

```
type(process_ptr_t) :: process_ptr
type(process_instance_t), allocatable, target :: process_instance
type(event_sample_data_t) :: data
class(eio_t), allocatable :: eio
type(string_t) :: sample
integer :: u_file, iostat
character(80) :: buffer

write (u, "(A)")  "* Test output: eio_ascii_3"
write (u, "(A)")  "*   Purpose: generate an event in ASCII debug format"
write (u, "(A)")  "*      and write weight to file"
write (u, "(A)")

call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_3"

allocate (eio_ascii_debug_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
```

2679

```
        call eio%final ()

        write (u, "(A)")
        write (u, "(A)")  "* File contents:"
        write (u, "(A)")

        u_file = free_unit ()
        open (u_file, file = char (sample // ".debug"), &
             action = "read", status = "old")
        do
           read (u_file, "(A)", iostat = iostat)  buffer
           if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
           if (iostat /= 0)  exit
           write (u, "(A)") trim (buffer)
        end do
        close (u_file)

        write (u, "(A)")
        write (u, "(A)")  "* Reset data"
        write (u, "(A)")

        deallocate (eio)
        allocate (eio_ascii_debug_t :: eio)

        select type (eio)
        type is (eio_ascii_debug_t)
           call eio%set_parameters (keep_beams = .true.)
        end select
        call eio%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call event%final ()
        deallocate (event)

        call cleanup_test_process (process, process_instance)
        deallocate (process_instance)
        deallocate (process)

        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: eio_ascii_3"

      end subroutine eio_ascii_3
```

We test the implementation of all I/O methods, method `hepevt`:

⟨*EIO ascii: execute tests*⟩+≡

```
    call test (eio_ascii_4, "eio_ascii_4", &
         "read and write event contents, format [hepevt]", &
         u, results)
```

⟨*EIO ascii: tests*⟩+≡

```
subroutine eio_ascii_4 (u)
  integer, intent(in) :: u
  type(model_list_t) :: model_list
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_ptr_t) :: process_ptr
  type(process_instance_t), allocatable, target :: process_instance
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat
  character(80) :: buffer

  write (u, "(A)")  "* Test output: eio_ascii_4"
  write (u, "(A)")  "*   Purpose: generate an event in ASCII hepevt format"
  write (u, "(A)")  "*      and write weight to file"
  write (u, "(A)")

  call syntax_model_file_init ()

  write (u, "(A)")  "* Initialize test process"

  allocate (process)
  process_ptr%ptr => process
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model_list)
  call process_instance%setup_event_data ()

  allocate (event)
  call event%basic_init ()
  call event%connect (process_instance, process%get_model_ptr ())

  call data%init (1)
  data%n_evt = 1
  data%n_beam = 2
  data%pdg_beam = 25
  data%energy_beam = 500
  data%proc_num_id = [42]
  data%cross_section(1) = 100
  data%error(1) = 1
  data%total_cross_section = sum (data%cross_section)

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_ascii_4"

  allocate (eio_ascii_hepevt_t :: eio)

  select type (eio)
  class is (eio_ascii_t);  call eio%set_parameters ()
  end select
```

```
      call eio%init_out (sample, [process_ptr], data)
      call event%generate (1, [0._default, 0._default])
      call event%evaluate_expressions ()

      call eio%output (event, i_prc = 1)
      call eio%write (u)
      call eio%final ()

    write (u, "(A)")
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // ".hepevt"), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_hepevt_t :: eio)

    select type (eio)
    type is (eio_ascii_hepevt_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_4"

  end subroutine eio_ascii_4
```

We test the implementation of all I/O methods, method `lha` (old LHA):

⟨*EIO ascii: execute tests*⟩+≡

```
    call test (eio_ascii_5, "eio_ascii_5", &
        "read and write event contents, format [lha]", &
        u, results)
```

⟨*EIO ascii: tests*⟩+≡

```
    subroutine eio_ascii_5 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_5"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII LHA format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
      process_ptr%ptr => process
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%pdg_beam = 25
      data%energy_beam = 500
      data%proc_num_id = [42]
      data%cross_section(1) = 100
      data%error(1) = 1
      data%total_cross_section = sum (data%cross_section)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_ascii_5"
```

```
allocate (eio_ascii_lha_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".lha"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_ascii_lha_t :: eio)

select type (eio)
type is (eio_ascii_lha_t)
   call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

call model_list%final ()
call syntax_model_file_final ()
```

2684

```
      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_ascii_5"

   end subroutine eio_ascii_5
```

We test the implementation of all I/O methods, method `long`:

⟨*EIO ascii: execute tests*⟩+≡
```
      call test (eio_ascii_6, "eio_ascii_6", &
           "read and write event contents, format [long]", &
           u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
   subroutine eio_ascii_6 (u)
     integer, intent(in) :: u
     type(model_list_t) :: model_list
     type(event_t), allocatable, target :: event
     type(process_t), allocatable, target :: process
     type(process_ptr_t) :: process_ptr
     type(process_instance_t), allocatable, target :: process_instance
     type(event_sample_data_t) :: data
     class(eio_t), allocatable :: eio
     type(string_t) :: sample
     integer :: u_file, iostat
     character(80) :: buffer

     write (u, "(A)")  "* Test output: eio_ascii_6"
     write (u, "(A)")  "*   Purpose: generate an event in ASCII long format"
     write (u, "(A)")  "*      and write weight to file"
     write (u, "(A)")

     call syntax_model_file_init ()

     write (u, "(A)")  "* Initialize test process"

     allocate (process)
     process_ptr%ptr => process
     allocate (process_instance)
     call prepare_test_process (process, process_instance, model_list)
     call process_instance%setup_event_data ()

     allocate (event)
     call event%basic_init ()
     call event%connect (process_instance, process%get_model_ptr ())

     call data%init (1)
     data%n_evt = 1
     data%n_beam = 2
     data%pdg_beam = 25
     data%energy_beam = 500
     data%proc_num_id = [42]
     data%cross_section(1) = 100
     data%error(1) = 1
     data%total_cross_section = sum (data%cross_section)
```

2685

```
write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_6"

allocate (eio_ascii_long_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".long.evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_ascii_long_t :: eio)

select type (eio)
type is (eio_ascii_long_t)
   call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
```

```
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_ascii_6"

    end subroutine eio_ascii_6
```

We test the implementation of all I/O methods, method `mokka`:

⟨*EIO ascii: execute tests*⟩+≡
```
      call test (eio_ascii_7, "eio_ascii_7", &
          "read and write event contents, format [mokka]", &
          u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
    subroutine eio_ascii_7 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_7"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII mokka format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
      process_ptr%ptr => process
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%pdg_beam = 25
```

2687

```
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_7"

allocate (eio_ascii_mokka_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".mokka.evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_ascii_mokka_t :: eio)

select type (eio)
type is (eio_ascii_mokka_t)
   call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
```

```
        write (u, "(A)")  "* Cleanup"

        call event%final ()
        deallocate (event)

        call cleanup_test_process (process, process_instance)
        deallocate (process_instance)
        deallocate (process)

        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: eio_ascii_7"

      end subroutine eio_ascii_7
```

We test the implementation of all I/O methods, method `short`:

⟨*EIO ascii: execute tests*⟩+≡
```
      call test (eio_ascii_8, "eio_ascii_8", &
           "read and write event contents, format [short]", &
           u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
      subroutine eio_ascii_8 (u)
        integer, intent(in) :: u
        type(model_list_t) :: model_list
        type(event_t), allocatable, target :: event
        type(process_t), allocatable, target :: process
        type(process_ptr_t) :: process_ptr
        type(process_instance_t), allocatable, target :: process_instance
        type(event_sample_data_t) :: data
        class(eio_t), allocatable :: eio
        type(string_t) :: sample
        integer :: u_file, iostat
        character(80) :: buffer

        write (u, "(A)")  "* Test output: eio_ascii_8"
        write (u, "(A)")  "*   Purpose: generate an event in ASCII short format"
        write (u, "(A)")  "*       and write weight to file"
        write (u, "(A)")

        call syntax_model_file_init ()

        write (u, "(A)")  "* Initialize test process"

        allocate (process)
        process_ptr%ptr => process
        allocate (process_instance)
        call prepare_test_process (process, process_instance, model_list)
        call process_instance%setup_event_data ()

        allocate (event)
        call event%basic_init ()
```

2689

```
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_8"

allocate (eio_ascii_short_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".short.evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_ascii_short_t :: eio)

select type (eio)
```

```
        type is (eio_ascii_short_t)
           call eio%set_parameters (keep_beams = .true.)
        end select
        call eio%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call event%final ()
        deallocate (event)

        call cleanup_test_process (process, process_instance)
        deallocate (process_instance)
        deallocate (process)

        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: eio_ascii_8"

      end subroutine eio_ascii_8
```

We test the implementation of all I/O methods, method `lha` (old LHA) in verbose version:

⟨*EIO ascii: execute tests*⟩+≡
```
    call test (eio_ascii_9, "eio_ascii_9", &
         "read and write event contents, format [lha_verb]", &
         u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
    subroutine eio_ascii_9 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_9"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII LHA verbose format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
```

```
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_9"

allocate (eio_ascii_lha_verb_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".lha.verb"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
```

2692

```
      write (u, "(A)")  "* Reset data"
      write (u, "(A)")

      deallocate (eio)
      allocate (eio_ascii_lha_verb_t :: eio)

      select type (eio)
      type is (eio_ascii_lha_verb_t)
         call eio%set_parameters (keep_beams = .true.)
      end select
      call eio%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_ascii_9"

    end subroutine eio_ascii_9
```

We test the implementation of all I/O methods, method `hepevt_verb`:

⟨*EIO ascii: execute tests*⟩+≡
```
    call test (eio_ascii_10, "eio_ascii_10", &
         "read and write event contents, format [hepevt_verb]", &
         u, results)
```

⟨*EIO ascii: tests*⟩+≡
```
    subroutine eio_ascii_10 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_10"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII hepevt verbose format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")
```

2693

```
call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_10"

allocate (eio_ascii_hepevt_verb_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".hepevt.verb"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
```

2694

```
      if (iostat /= 0)  exit
      write (u, "(A)") trim (buffer)
   end do
   close (u_file)

   write (u, "(A)")
   write (u, "(A)")  "* Reset data"
   write (u, "(A)")

   deallocate (eio)
   allocate (eio_ascii_hepevt_verb_t :: eio)

   select type (eio)
   type is (eio_ascii_hepevt_verb_t)
      call eio%set_parameters (keep_beams = .true.)
   end select
   call eio%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call event%final ()
   deallocate (event)

   call cleanup_test_process (process, process_instance)
   deallocate (process_instance)
   deallocate (process)

   call model_list%final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: eio_ascii_10"

 end subroutine eio_ascii_10
```

19.11   STDHEP File Formats

Here, we implement the two existing STDHEP file formats, one based on the
HEPRUP/HEPEUP common blocks, the other based on the HEPEVT common
block. The second one is actually the standard STDHEP format.

⟨eio_stdhep.f90⟩≡
  ⟨*File header*⟩

  module eio_stdhep

    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use diagnostics !NODEP!
    use unit_tests
```

```
      use lorentz !NODEP!
      use models
      use particles
      use beams
      use processes
      use events
      use eio_data
      use eio_base
      use hep_common
      use hep_events
```

⟨*Standard module head*⟩

⟨*EIO stdhep: public*⟩

⟨*EIO stdhep: types*⟩

⟨*EIO stdhep: variables*⟩

```
contains
```

⟨*EIO stdhep: procedures*⟩

⟨*EIO stdhep: tests*⟩

```
end module eio_stdhep
```

19.11.1   Type

⟨*EIO stdhep: public*⟩≡
```
      public :: eio_stdhep_t
```

⟨*EIO stdhep: types*⟩≡
```
      type, abstract, extends (eio_t) :: eio_stdhep_t
         logical :: writing = .false.
         integer :: unit = 0
         logical :: keep_beams = .false.
         integer(i64) :: n_events_expected = 0
       contains
       ⟨EIO stdhep: eio stdhep: TBP⟩
      end type eio_stdhep_t
```

⟨*EIO stdhep: public*⟩+≡
```
      public :: eio_stdhep_hepevt_t
```

⟨*EIO stdhep: types*⟩+≡
```
      type, extends (eio_stdhep_t) :: eio_stdhep_hepevt_t
      end type eio_stdhep_hepevt_t
```

⟨*EIO stdhep: public*⟩+≡
```
      public :: eio_stdhep_hepeup_t
```

⟨*EIO stdhep: types*⟩+≡
```
type, extends (eio_stdhep_t) :: eio_stdhep_hepeup_t
end type eio_stdhep_hepeup_t
```

19.11.2   Specific Methods

Set parameters that are specifically used with STDHEP file formats.

⟨*EIO stdhep: eio stdhep: TBP*⟩≡
```
procedure :: set_parameters => eio_stdhep_set_parameters
```

⟨*EIO stdhep: procedures*⟩≡
```
subroutine eio_stdhep_set_parameters (eio, keep_beams, extension)
  class(eio_stdhep_t), intent(inout) :: eio
  logical, intent(in), optional :: keep_beams
  type(string_t), intent(in), optional :: extension
  if (present (keep_beams))  eio%keep_beams = keep_beams
  if (present (extension)) then
     eio%extension = extension
  else
     select type (eio)
     type is (eio_stdhep_hepevt_t)
        eio%extension = "stdhep"
     type is (eio_stdhep_hepeup_t)
        eio%extension = "up.stdhep"
     end select
  end if
end subroutine eio_stdhep_set_parameters
```

19.11.3   Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
procedure :: write => eio_stdhep_write
```

⟨*EIO stdhep: procedures*⟩+≡
```
subroutine eio_stdhep_write (object, unit)
  class(eio_stdhep_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = output_unit (unit)
  write (u, "(1x,A)")  "STDHEP event stream:"
  if (object%writing) then
     write (u, "(3x,A,A)")  "Writing to file  = ", char (object%filename)
  else
     write (u, "(3x,A)")  "[closed]"
  end if
  write (u, "(3x,A,L1)")    "Keep beams       = ", object%keep_beams
end subroutine eio_stdhep_write
```

Finalizer: close any open file.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
    procedure :: final => eio_stdhep_final
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine eio_stdhep_final (object)
      class(eio_stdhep_t), intent(inout) :: object
      if (object%writing) then
         write (msg_buffer, "(A,A,A)")  "Events: closing STDHEP file '", &
               char (object%filename), "'"
         call msg_message ()
         call stdhep_end
         object%writing = .false.
      end if
    end subroutine eio_stdhep_final
```

Split event file: increment the counter, close the current file, open a new one. If
the file needs a header, repeat it for the new file. (We assume that the common
block contents are still intact.)

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
    procedure :: split_out => eio_stdhep_split_out
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine eio_stdhep_split_out (eio)
      class(eio_stdhep_t), intent(inout) :: eio
      if (eio%split) then
         eio%split_index = eio%split_index + 1
         call eio%set_filename ()
         write (msg_buffer, "(A,A,A)")  "Events: writing to STDHEP file '", &
               char (eio%filename), "'"
         call msg_message ()
         call stdhep_end
         select type (eio)
         type is (eio_stdhep_hepeup_t)
            call stdhep_init (char (eio%filename), &
                  "WHIZARD event sample", eio%n_events_expected)
            call stdhep_write (STDHEP_HEPRUP)
         type is (eio_stdhep_hepevt_t)
            call stdhep_init (char (eio%filename), &
                  "WHIZARD event sample", eio%n_events_expected)
         end select
      end if
    end subroutine eio_stdhep_split_out
```

Initialize event writing.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
    procedure :: init_out => eio_stdhep_init_out
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine eio_stdhep_init_out (eio, sample, process_ptr, data, success)
      class(eio_stdhep_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(in), optional :: data
```

```
      logical, intent(out), optional :: success
      integer :: i
      if (.not. present (data)) &
          call msg_bug ("STDHEP initialization: missing data")
      if (data%n_beam /= 2) &
          call msg_fatal ("STDHEP: defined for scattering processes only")
      call eio%set_splitting (data)
      eio%sample = sample
      select type (eio)
      type is (eio_stdhep_hepevt_t)
         eio%extension = "stdhep"
      type is (eio_stdhep_hepeup_t)
         eio%extension = "up.stdhep"
      end select
      call eio%set_filename ()
      eio%n_events_expected = data%n_evt
      eio%unit = free_unit ()
      write (msg_buffer, "(A,A,A)")  "Events: writing to STDHEP file '", &
          char (eio%filename), "'"
      call msg_message ()
      eio%writing = .true.
      select type (eio)
      type is (eio_stdhep_hepeup_t)
         call heprup_init &
             (data%pdg_beam, &
             data%energy_beam, &
             n_processes = data%n_proc, &
             unweighted = data%unweighted, &
             negative_weights = data%negative_weights)
         do i = 1, data%n_proc
           call heprup_set_process_parameters (i = i, &
                process_id = data%proc_num_id(i), &
                cross_section = data%cross_section(i), &
                error = data%error(i))
         end do
         call stdhep_init (char (eio%filename), &
             "WHIZARD event sample", eio%n_events_expected)
         call stdhep_write (STDHEP_HEPRUP)
      type is (eio_stdhep_hepevt_t)
         call stdhep_init (char (eio%filename), &
             "WHIZARD event sample", eio%n_events_expected)
      end select
      if (present (success))  success = .true.
    end subroutine eio_stdhep_init_out
```

Initialize event reading.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
    procedure :: init_in => eio_stdhep_init_in
```

⟨*EIO stdhep: procedures*⟩+≡

```
    subroutine eio_stdhep_init_in &
        (eio, sample, process_ptr, data, success, extension)
      class(eio_stdhep_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
```

```
      type(string_t), intent(in), optional :: extension
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      call msg_bug ("STDHEP: event input not supported")
      if (present (success))  success = .false.
    end subroutine eio_stdhep_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
    procedure :: switch_inout => eio_stdhep_switch_inout
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine eio_stdhep_switch_inout (eio, success)
      class(eio_stdhep_t), intent(inout) :: eio
      logical, intent(out), optional :: success
      call msg_bug ("STDHEP: in-out switch not supported")
      if (present (success))  success = .false.
    end subroutine eio_stdhep_switch_inout
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
    procedure :: output => eio_stdhep_output
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine eio_stdhep_output (eio, event, i_prc, reading)
      class(eio_stdhep_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      integer, intent(in) :: i_prc
      logical, intent(in), optional :: reading
      if (eio%writing) then
        select type (eio)
        type is (eio_stdhep_hepeup_t)
          call hepeup_from_event (event, &
               process_index = i_prc, &
               keep_beams = eio%keep_beams)
          call stdhep_write (STDHEP_HEPEUP)
        type is (eio_stdhep_hepevt_t)
          call hepevt_from_event (event, &
               i_evt = event%expr%index, &
               keep_beams = eio%keep_beams)
          call stdhep_write (STDHEP_HEPEVT)
        end select
      else
        call eio%write ()
        call msg_fatal ("STDHEP file is not open for writing")
      end if
    end subroutine eio_stdhep_output
```

Input an event.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡
```
    procedure :: input_i_prc => eio_stdhep_input_i_prc
    procedure :: input_event => eio_stdhep_input_event
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine eio_stdhep_input_i_prc (eio, i_prc, iostat)
      class(eio_stdhep_t), intent(inout) :: eio
      integer, intent(out) :: i_prc
      integer, intent(out) :: iostat
      call msg_bug ("STDHEP: event input not supported")
      i_prc = 0
      iostat = 1
    end subroutine eio_stdhep_input_i_prc

    subroutine eio_stdhep_input_event (eio, event, iostat)
      class(eio_stdhep_t), intent(inout) :: eio
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      call msg_bug ("STDHEP: event input not supported")
      iostat = 1
    end subroutine eio_stdhep_input_event
```

STDHEP speficic routines.

⟨*EIO stdhep: public*⟩+≡
```
    public :: stdhep_init
    public :: stdhep_write
    public :: stdhep_end
```

⟨*EIO stdhep: procedures*⟩+≡
```
    subroutine stdhep_init (file, title, nevt)
      character(len=*), intent(in) :: file, title
      integer(i64), intent(in) :: nevt
      integer(i32) :: nevt32
      external stdxwinit, stdxwrt
      nevt32 = min (nevt, int (huge (1_i32), i64))
      call stdxwinit (file, title, nevt32, istr, lok)
      call stdxwrt (100, istr, lok)
    end subroutine stdhep_init

    subroutine stdhep_write (ilbl)
      integer, intent(in) :: ilbl
      external stdxwrt
      call stdxwrt (ilbl, istr, lok)
    end subroutine stdhep_write

    subroutine stdhep_end
      external stdxend
      call stdxend (istr)
    end subroutine stdhep_end
```

19.11.4 Variables

⟨*EIO stdhep: variables*⟩≡
```
    integer, save :: istr, lok
    integer, parameter :: &
         STDHEP_HEPEVT = 1, STDHEP_HEPEUP = 11, STDHEP_HEPRUP = 12
```

19.11.5 Unit tests

⟨*EIO stdhep: public*⟩+≡
```
public :: eio_stdhep_test
```

⟨*EIO stdhep: tests*⟩≡
```
subroutine eio_stdhep_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨EIO stdhep: execute tests⟩
end subroutine eio_stdhep_test
```

Test I/O methods

We test the implementation of the STDHEP HEPEVT I/O method:

⟨*EIO stdhep: execute tests*⟩≡
```
call test (eio_stdhep_1, "eio_stdhep_1", &
    "read and write event contents, format [stdhep]", &
    u, results)
```

⟨*EIO stdhep: tests*⟩+≡
```
subroutine eio_stdhep_1 (u)
  integer, intent(in) :: u
  type(model_list_t) :: model_list
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_ptr_t) :: process_ptr
  type(process_instance_t), allocatable, target :: process_instance
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat
  character(80) :: buffer

  write (u, "(A)")  "* Test output: eio_stdhep_1"
  write (u, "(A)")  "*   Purpose: generate an event in STDHEP HEPEVT format"
  write (u, "(A)")  "*      and write weight to file"
  write (u, "(A)")

  call syntax_model_file_init ()

  write (u, "(A)")  "* Initialize test process"

  allocate (process)
  process_ptr%ptr => process
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model_list)
  call process_instance%setup_event_data ()

  allocate (event)
  call event%basic_init ()
  call event%connect (process_instance, process%get_model_ptr ())

  call data%init (1)
```

```
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_stdhep_1"

allocate (eio_stdhep_hepevt_t :: eio)

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".stdhep"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
end do
!!! JRR: WK please check: should be replaced by a XDR reader? (#530)
!  write (u, "(A)") trim (buffer)
!  The number of lines is system-dependent!
write (u, "(A)") "    Successfully read STDHEP HEPEVT file"
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_stdhep_hepevt_t :: eio)

select type (eio)
type is (eio_stdhep_hepevt_t)
   call eio%set_parameters (keep_beams = .true.)
end select
```

```
        call eio%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call event%final ()
        deallocate (event)

        call cleanup_test_process (process, process_instance)
        deallocate (process_instance)
        deallocate (process)

        call model_list%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: eio_stdhep_1"

      end subroutine eio_stdhep_1
```

We test the implementation of the STDHEP HEPEUP I/O method:

⟨*EIO stdhep: execute tests*⟩+≡
```
    call test (eio_stdhep_2, "eio_stdhep_2", &
         "read and write event contents, format [stdhep]", &
         u, results)
```

⟨*EIO stdhep: tests*⟩+≡
```
    subroutine eio_stdhep_2 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_stdhep_2"
      write (u, "(A)")  "*   Purpose: generate an event in STDHEP HEPEUP format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
      process_ptr%ptr => process
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()
```

2704

```
allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_stdhep_2"

allocate (eio_stdhep_hepeup_t :: eio)

call eio%init_out (sample, [process_ptr], data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char(sample // ".up.stdhep"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
end do
!!! JRR: WK please check: should be replaced by a XDR reader? (#530)
!  write (u, "(A)") trim (buffer)
!  The number of lines is system-dependent!
write (u, "(A)") "     Successfully read STDHEP HEPEUP file"
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
```

```
      allocate (eio_stdhep_hepeup_t :: eio)

      select type (eio)
      type is (eio_stdhep_hepeup_t)
         call eio%set_parameters (keep_beams = .true.)
      end select
      call eio%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_stdhep_2"

    end subroutine eio_stdhep_2
```

19.12   Event Weight Output

This is an output-only format. For each event, we print the indices that identify
process, process part (MCI group), and term. As numerical information we print
the squared matrix element (trace) and the event weight.

⟨eio_weights.f90⟩≡
  ⟨*File header*⟩

  module eio_weights

    use kinds !NODEP!
  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use diagnostics !NODEP!
    use unit_tests

    use lorentz !NODEP!
    use variables
    use models
    use particles
    use beams
    use processes
    use events
    use eio_data
    use eio_base

```

⟨*Standard module head*⟩

⟨*EIO weights: public*⟩

⟨*EIO weights: types*⟩

```
contains
```

⟨*EIO weights: procedures*⟩

⟨*EIO weights: tests*⟩

```
end module eio_weights
```

19.12.1 Type

⟨*EIO weights: public*⟩≡
```
    public :: eio_weights_t
```

⟨*EIO weights: types*⟩≡
```
    type, extends (eio_t) :: eio_weights_t
      logical :: writing = .false.
      integer :: unit = 0
      logical :: pacify = .false.
     contains
     ⟨EIO weights: eio weights: TBP⟩
    end type eio_weights_t
```

19.12.2 Specific Methods

Set pacify flags.

⟨*EIO weights: eio weights: TBP*⟩≡
```
    procedure :: set_parameters => eio_weights_set_parameters
```

⟨*EIO weights: procedures*⟩≡
```
    subroutine eio_weights_set_parameters (eio, pacify)
      class(eio_weights_t), intent(inout) :: eio
      logical, intent(in), optional :: pacify
      if (present (pacify))  eio%pacify = pacify
    end subroutine eio_weights_set_parameters
```

19.12.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
    procedure :: write => eio_weights_write
```

⟨*EIO weights: procedures*⟩+≡
```
    subroutine eio_weights_write (object, unit)
      class(eio_weights_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "Weight stream:"
      if (object%writing) then
         write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
         write (u, "(3x,A,L1)") "Reduced I/O prec. = ", object%pacify
      else
         write (u, "(3x,A)")  "[closed]"
      end if
    end subroutine eio_weights_write
```

Finalizer: close any open file.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
    procedure :: final => eio_weights_final
```

⟨*EIO weights: procedures*⟩+≡
```
    subroutine eio_weights_final (object)
      class(eio_weights_t), intent(inout) :: object
      if (object%writing) then
         write (msg_buffer, "(A,A,A)")  "Events: closing weight stream file '", &
               char (object%filename), "'"
         call msg_message ()
         close (object%unit)
         object%writing = .false.
      end if
    end subroutine eio_weights_final
```

Initialize event writing.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
    procedure :: init_out => eio_weights_init_out
```

⟨*EIO weights: procedures*⟩+≡
```
    subroutine eio_weights_init_out &
         (eio, sample, process_ptr, data, success)
      class(eio_weights_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(in), optional :: data
      logical, intent(out), optional :: success
      eio%filename = sample // ".weights.dat"
      eio%unit = free_unit ()
      write (msg_buffer, "(A,A,A)")  "Events: writing to weight stream file '", &
            char (eio%filename), "'"
      call msg_message ()
      eio%writing = .true.
      open (eio%unit, file = char (eio%filename), &
            action = "write", status = "replace")
      if (present (success))  success = .true.
    end subroutine eio_weights_init_out
```

Initialize event reading.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
    procedure :: init_in => eio_weights_init_in
```

⟨*EIO weights: procedures*⟩+≡
```
    subroutine eio_weights_init_in &
        (eio, sample, process_ptr, data, success, extension)
      class(eio_weights_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(process_ptr_t), dimension(:), intent(in) :: process_ptr
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      call msg_bug ("Weight stream: event input not supported")
      if (present (success))  success = .false.
    end subroutine eio_weights_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
    procedure :: switch_inout => eio_weights_switch_inout
```

⟨*EIO weights: procedures*⟩+≡
```
    subroutine eio_weights_switch_inout (eio, success)
      class(eio_weights_t), intent(inout) :: eio
      logical, intent(out), optional :: success
      call msg_bug ("Weight stream: in-out switch not supported")
      if (present (success))  success = .false.
    end subroutine eio_weights_switch_inout
```

Output an event. Write first the event indices, then weight and two values of the squared matrix element: `sqme_ref` is the value stored in the event record, and `sqme_prc` is the one stored in the process instance. (They can differ: when recalculating, the former is read from file and the latter is the result of the new calculation.)

For the alternative entries, the `sqme` value is always obtained by a new calculation, and thus qualifies as `sqme_prc`.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
    procedure :: output => eio_weights_output
```

⟨*EIO weights: procedures*⟩+≡
```
    subroutine eio_weights_output (eio, event, i_prc, reading)
      class(eio_weights_t), intent(inout) :: eio
      type(event_t), intent(in), target :: event
      integer, intent(in) :: i_prc
      logical, intent(in), optional :: reading
      integer :: i_mci, i_term, n_alt, i
      real(default) :: weight, sqme_ref, sqme_prc
      if (eio%writing) then
         i_mci = event%get_i_mci ()
         i_term = event%get_i_term ()
         weight = event%weight_prc
         sqme_ref = event%sqme_ref
         sqme_prc = event%sqme_prc
         n_alt = event%n_alt
```

```
1       format (I0,3(1x,ES17.10),3(1x,I0))
2       format (I0,3(1x,ES15.8),3(1x,I0))
        if (eio%pacify) then
           write (eio%unit, 2)  0, weight, sqme_prc, sqme_ref, &
                i_prc, i_mci, i_term
        else
           write (eio%unit, 1)  0, weight, sqme_prc, sqme_ref, &
                i_prc, i_mci, i_term
        end if
        do i = 1, n_alt
           weight = event%weight_alt(i)
           sqme_prc = event%sqme_alt(i)
           if (eio%pacify) then
              write (eio%unit, 2)  i, weight, sqme_prc
           else
              write (eio%unit, 1)  i, weight, sqme_prc
           end if
        end do
     else
        call eio%write ()
        call msg_fatal ("Weight stream file is not open for writing")
     end if
  end subroutine eio_weights_output
```

Input an event.

⟨EIO weights: eio weights: TBP⟩+≡
```
    procedure :: input_i_prc => eio_weights_input_i_prc
    procedure :: input_event => eio_weights_input_event
```

⟨EIO weights: procedures⟩+≡
```
    subroutine eio_weights_input_i_prc (eio, i_prc, iostat)
      class(eio_weights_t), intent(inout) :: eio
      integer, intent(out) :: i_prc
      integer, intent(out) :: iostat
      call msg_bug ("Weight stream: event input not supported")
      i_prc = 0
      iostat = 1
    end subroutine eio_weights_input_i_prc

    subroutine eio_weights_input_event (eio, event, iostat)
      class(eio_weights_t), intent(inout) :: eio
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      call msg_bug ("Weight stream: event input not supported")
      iostat = 1
    end subroutine eio_weights_input_event
```

19.12.4 Unit tests

⟨EIO weights: public⟩+≡
```
    public :: eio_weights_test
```

⟨*EIO weights: tests*⟩≡
```
    subroutine eio_weights_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨EIO weights: execute tests⟩
    end subroutine eio_weights_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO weights: execute tests*⟩≡
```
    call test (eio_weights_1, "eio_weights_1", &
        "read and write event contents", &
        u, results)
```

⟨*EIO weights: tests*⟩+≡
```
    subroutine eio_weights_1 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(process_instance_t), allocatable, target :: process_instance
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_weights_1"
      write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
      write (u, "(A)")

      call syntax_model_file_init ()

      write (u, "(A)")  "* Initialize test process"

      allocate (process)
      process_ptr%ptr => process
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model_list)
      call process_instance%setup_event_data ()

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_weights_1"

      allocate (eio_weights_t :: eio)
```

2711

```
      call eio%init_out (sample, [process_ptr])
      call event%generate (1, [0._default, 0._default])

      call eio%output (event, i_prc = 42)
      call eio%write (u)
      call eio%final ()

      write (u, "(A)")
      write (u, "(A)")  "* File contents: &
          &(weight, sqme(evt), sqme(prc), i_prc, i_mci, i_term)"
      write (u, "(A)")

      u_file = free_unit ()
      open (u_file, file = "eio_weights_1.weights.dat", &
          action = "read", status = "old")
      read (u_file, "(A)")  buffer
      write (u, "(A)") trim (buffer)
      close (u_file)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call event%final ()
      deallocate (event)

      call cleanup_test_process (process, process_instance)
      deallocate (process_instance)
      deallocate (process)

      call model_list%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: eio_weights_1"

    end subroutine eio_weights_1
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO weights: execute tests*⟩+≡
```
    call test (eio_weights_2, "eio_weights_2", &
        "multiple weights", &
        u, results)
```

⟨*EIO weights: tests*⟩+≡
```
    subroutine eio_weights_2 (u)
      integer, intent(in) :: u
      type(model_list_t) :: model_list
      type(var_list_t) :: var_list
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
```

```
type(process_ptr_t) :: process_ptr
type(process_instance_t), allocatable, target :: process_instance
class(eio_t), allocatable :: eio
type(string_t) :: sample
integer :: u_file, i
character(80) :: buffer

write (u, "(A)")  "* Test output: eio_weights_2"
write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
write (u, "(A)")

call syntax_model_file_init ()

write (u, "(A)")  "* Initialize test process"

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, model_list)
call process_instance%setup_event_data ()

call var_list_append_log (var_list, var_str ("?unweighted"), .false., &
    intrinsic = .true.)
call var_list_append_string (var_list, var_str ("$sample_normalization"), &
    var_str ("auto"), intrinsic = .true.)

allocate (event)
call event%basic_init (var_list, n_alt = 2)
call event%connect (process_instance, process%get_model_ptr ())

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_weights_2"

allocate (eio_weights_t :: eio)

call eio%init_out (sample, [process_ptr])
select type (eio)
type is (eio_weights_t)
   call eio%set_parameters (pacify = .true.)
end select
call event%generate (1, [0._default, 0._default])
call event%set (sqme_alt = [2._default, 3._default])
call event%set (weight_alt = [2 * event%weight_prc, 3 * event%weight_prc])

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents: &
    &(weight, sqme(evt), sqme(prc), i_prc, i_mci, i_term)"
```

2713

```
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = "eio_weights_2.weights.dat", &
         action = "read", status = "old")
    do i = 1, 3
       read (u_file, "(A)")  buffer
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_weights_2"

  end subroutine eio_weights_2
```

Chapter 20

The SUSY Les Houches Accord

The SUSY Les Houches Accord defines a standard interfaces for storing the physics data of SUSY models. Here, we provide the means for reading, storing, and writing such data.

⟨slha_interface.f90⟩≡
  ⟨*File header*⟩

  module slha_interface

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use limits, only: EOF, VERSION_STRING !NODEP!
    use constants !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use unit_tests
    use os_interface
    use ifiles
    use lexers
    use syntax_rules
    use parser
    use variables
    use expressions
    use models

  ⟨*Standard module head*⟩

  ⟨*SLHA: public*⟩

  ⟨*SLHA: parameters*⟩

  ⟨*SLHA: variables*⟩

    save

  contains

⟨*SLHA: procedures*⟩

⟨*SLHA: tests*⟩

```
end module slha_interface
```

20.0.5   Preprocessor

SLHA is a mixed-format standard. It should be read in assuming free format (but line-oriented), but it has some fixed-format elements.

To overcome this difficulty, we implement a preprocessing step which transforms the SLHA into a format that can be swallowed by our generic free-format lexer and parser. Each line with a blank first character is assumed to be a data line. We prepend a 'DATA' keyword to these lines. Furthermore, to enforce line-orientation, each line is appended a '$' key which is recognized by the parser. To do this properly, we first remove trailing comments, and skip lines consisting only of comments.

The preprocessor reads from a stream and puts out an `ifile`. Blocks that are not recognized are skipped. For some blocks, data items are quoted, so they can be read as strings if necessary.

⟨*SLHA: parameters*⟩≡

```
    integer, parameter :: MODE_SKIP = 0, MODE_DATA = 1, MODE_INFO = 2
```

⟨*SLHA: procedures*⟩≡

```
  subroutine slha_preprocess (stream, ifile)
    type(stream_t), intent(inout), target :: stream
    type(ifile_t), intent(out) :: ifile
    type(string_t) :: buffer, line, item
    integer :: iostat
    integer :: mode
    mode = MODE
    SCAN_FILE: do
       call stream_get_record (stream, buffer, iostat)
       select case (iostat)
       case (0)
          call split (buffer, line, "#")
          if (len_trim (line) == 0)  cycle SCAN_FILE
          select case (char (extract (line, 1, 1)))
          case ("B", "b")
             mode = check_block_handling (line)
             call ifile_append (ifile, line // "$")
          case ("D", "d")
             mode = MODE_DATA
             call ifile_append (ifile, line // "$")
          case (" ")
             select case (mode)
             case (MODE_DATA)
                call ifile_append (ifile, "DATA" // line // "$")
             case (MODE_INFO)
                line = adjustl (line)
                call split (line, item, " ")
```

```
              call ifile_append (ifile, "INFO" // " " // item // " " &
                   // '"' // trim (adjustl (line)) // '" $')
           end select
        case default
           call msg_message (char (line))
           call msg_fatal ("SLHA: Incomprehensible line")
        end select
     case (EOF)
        exit SCAN_FILE
     case default
        call msg_fatal ("SLHA: I/O error occured while reading SLHA input")
     end select
  end do SCAN_FILE
end subroutine slha_preprocess
```

Return the mode that we should treat this block with. We need to recognize only those blocks that we actually use.

⟨*SLHA: procedures*⟩+≡
```
function check_block_handling (line) result (mode)
  integer :: mode
  type(string_t), intent(in) :: line
  type(string_t) :: buffer, key, block_name
  buffer = trim (line)
  call split (buffer, key, " ")
  buffer = adjustl (buffer)
  call split (buffer, block_name, " ")
  block_name = trim (adjustl (upper_case (block_name)))
  select case (char (block_name))
  case ("MODSEL", "MINPAR", "SMINPUTS")
     mode = MODE_DATA
  case ("MASS")
     mode = MODE_DATA
  case ("NMIX", "UMIX", "VMIX", "STOPMIX", "SBOTMIX", "STAUMIX")
     mode = MODE_DATA
  case ("NMHMIX", "NMAMIX", "NMNMIX", "NMSSMRUN")
     mode = MODE_DATA
  case ("ALPHA", "HMIX")
     mode = MODE_DATA
  case ("AU", "AD", "AE")
     mode = MODE_DATA
  case ("SPINFO", "DCINFO")
     mode = MODE_INFO
  case default
     mode = MODE_SKIP
  end select
end function check_block_handling
```

20.0.6 Lexer and syntax

⟨*SLHA: variables*⟩≡
```
type(syntax_t), target :: syntax_slha
```

⟨*SLHA: public*⟩≡
```
    public :: syntax_slha_init
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine syntax_slha_init ()
      type(ifile_t) :: ifile
      call define_slha_syntax (ifile)
      call syntax_init (syntax_slha, ifile)
      call ifile_final (ifile)
    end subroutine syntax_slha_init
```

⟨*SLHA: public*⟩+≡
```
    public :: syntax_slha_final
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine syntax_slha_final ()
      call syntax_final (syntax_slha)
    end subroutine syntax_slha_final
```

⟨*SLHA: public*⟩+≡
```
    public :: syntax_slha_write
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine syntax_slha_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_slha, unit)
    end subroutine syntax_slha_write
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine define_slha_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ slha = chunk*")
      call ifile_append (ifile, "ALT chunk = block_def | decay_def")
      call ifile_append (ifile, "SEQ block_def = " &
           // "BLOCK block_spec '$' block_line*")
      call ifile_append (ifile, "KEY BLOCK")
      call ifile_append (ifile, "SEQ block_spec = block_name qvalue?")
      call ifile_append (ifile, "IDE block_name")
      call ifile_append (ifile, "SEQ qvalue = qname '=' real")
      call ifile_append (ifile, "IDE qname")
      call ifile_append (ifile, "KEY '='")
      call ifile_append (ifile, "REA real")
      call ifile_append (ifile, "KEY '$'")
      call ifile_append (ifile, "ALT block_line = block_data | block_info")
      call ifile_append (ifile, "SEQ block_data = DATA data_line '$'")
      call ifile_append (ifile, "KEY DATA")
      call ifile_append (ifile, "SEQ data_line = data_item+")
      call ifile_append (ifile, "ALT data_item = signed_number | number")
      call ifile_append (ifile, "SEQ signed_number = sign number")
      call ifile_append (ifile, "ALT sign = '+' | '-'")
      call ifile_append (ifile, "ALT number = integer | real")
      call ifile_append (ifile, "INT integer")
      call ifile_append (ifile, "KEY '-'")
      call ifile_append (ifile, "KEY '+'")
      call ifile_append (ifile, "SEQ block_info = INFO info_line '$'")
```

```
      call ifile_append (ifile, "KEY INFO")
      call ifile_append (ifile, "SEQ info_line = integer string_literal")
      call ifile_append (ifile, "QUO string_literal = '""'...'""'")
      call ifile_append (ifile, "SEQ decay_def = " &
           // "DECAY decay_spec '$' decay_data*")
      call ifile_append (ifile, "KEY DECAY")
      call ifile_append (ifile, "SEQ decay_spec = pdg_code data_item")
      call ifile_append (ifile, "ALT pdg_code = signed_integer | integer")
      call ifile_append (ifile, "SEQ signed_integer = sign integer")
      call ifile_append (ifile, "SEQ decay_data = DATA decay_line '$'")
      call ifile_append (ifile, "SEQ decay_line = data_item integer pdg_code+")
    end subroutine define_slha_syntax
```

The SLHA specification allows for string data items in certain places. Currently, we do not interpret them, but the strings, which are not quoted, must be parsed somehow. The hack for this problem is to allow essentially all characters as special characters, so the string can be read before it is discarded.

⟨*SLHA: public*⟩+≡
```
    public :: lexer_init_slha
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine lexer_init_slha (lexer)
      type(lexer_t), intent(out) :: lexer
      call lexer_init (lexer, &
           comment_chars = "#", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "+-=$", &
           special_class = [ "" ], &
           keyword_list = syntax_get_keyword_list_ptr (syntax_slha), &
           upper_case_keywords = .true.)  ! $
    end subroutine lexer_init_slha
```

20.0.7   Interpreter

Find blocks

From the parse tree, find the node that represents a particular block. If `required` is true, issue an error if not found. Since `block_name` is always invoked with capital letters, we have to capitalize `pn_block_name`.

⟨*SLHA: procedures*⟩+≡
```
    function slha_get_block_ptr &
         (parse_tree, block_name, required) result (pn_block)
      type(parse_node_t), pointer :: pn_block
      type(parse_tree_t), intent(in) :: parse_tree
      type(string_t), intent(in) :: block_name
      logical, intent(in) :: required
      type(parse_node_t), pointer :: pn_root, pn_block_spec, pn_block_name
      pn_root => parse_tree_get_root_ptr (parse_tree)
      pn_block => parse_node_get_sub_ptr (pn_root)
      do while (associated (pn_block))
         select case (char (parse_node_get_rule_key (pn_block)))
```

```
         case ("block_def")
            pn_block_spec => parse_node_get_sub_ptr (pn_block, 2)
            pn_block_name => parse_node_get_sub_ptr (pn_block_spec)
            if (trim (adjustl (upper_case (parse_node_get_string &
                     (pn_block_name)))) == block_name) then
               return
            end if
         end select
         pn_block => parse_node_get_next_ptr (pn_block)
      end do
      if (required) then
         call msg_fatal ("SLHA: block '" // char (block_name) // "' not found")
      end if
    end function slha_get_block_ptr
```

Scan the file for the first/next DECAY block.

⟨*SLHA: procedures*⟩+≡

```
    function slha_get_first_decay_ptr (parse_tree) result (pn_decay)
      type(parse_node_t), pointer :: pn_decay
      type(parse_tree_t), intent(in) :: parse_tree
      type(parse_node_t), pointer :: pn_root
      pn_root => parse_tree_get_root_ptr (parse_tree)
      pn_decay => parse_node_get_sub_ptr (pn_root)
      do while (associated (pn_decay))
         select case (char (parse_node_get_rule_key (pn_decay)))
         case ("decay_def")
            return
         end select
         pn_decay => parse_node_get_next_ptr (pn_decay)
      end do
    end function slha_get_first_decay_ptr

    function slha_get_next_decay_ptr (pn_block) result (pn_decay)
      type(parse_node_t), pointer :: pn_decay
      type(parse_node_t), intent(in), target :: pn_block
      pn_decay => parse_node_get_next_ptr (pn_block)
      do while (associated (pn_decay))
         select case (char (parse_node_get_rule_key (pn_decay)))
         case ("decay_def")
            return
         end select
         pn_decay => parse_node_get_next_ptr (pn_decay)
      end do
    end function slha_get_next_decay_ptr
```

Extract and transfer data from blocks

Given the parse node of a block, find the parse node of a particular switch or
data line. Return this node and the node of the data item following the integer
code.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_find_index_ptr (pn_block, pn_data, pn_item, code)
```

```
      type(parse_node_t), intent(in), target :: pn_block
      type(parse_node_t), intent(out), pointer :: pn_data
      type(parse_node_t), intent(out), pointer :: pn_item
      integer, intent(in) :: code
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      call slha_next_index_ptr (pn_data, pn_item, code)
    end subroutine slha_find_index_ptr

    subroutine slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
      type(parse_node_t), intent(in), target :: pn_block
      type(parse_node_t), intent(out), pointer :: pn_data
      type(parse_node_t), intent(out), pointer :: pn_item
      integer, intent(in) :: code1, code2
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      call slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
    end subroutine slha_find_index_pair_ptr
```

Starting from the pointer to a data line, find a data line with the given integer code.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_next_index_ptr (pn_data, pn_item, code)
      type(parse_node_t), intent(inout), pointer :: pn_data
      integer, intent(in) :: code
      type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_line, pn_code
      do while (associated (pn_data))
         pn_line => parse_node_get_sub_ptr (pn_data, 2)
         pn_code => parse_node_get_sub_ptr (pn_line)
         select case (char (parse_node_get_rule_key (pn_code)))
         case ("integer")
            if (parse_node_get_integer (pn_code) == code) then
               pn_item => parse_node_get_next_ptr (pn_code)
               return
            end if
         end select
         pn_data => parse_node_get_next_ptr (pn_data)
      end do
      pn_item => null ()
    end subroutine slha_next_index_ptr
```

Starting from the pointer to a data line, find a data line with the given integer code pair.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
      type(parse_node_t), intent(inout), pointer :: pn_data
      integer, intent(in) :: code1, code2
      type(parse_node_t), intent(out), pointer :: pn_item
      type(parse_node_t), pointer :: pn_line, pn_code1, pn_code2
      do while (associated (pn_data))
         pn_line => parse_node_get_sub_ptr (pn_data, 2)
         pn_code1 => parse_node_get_sub_ptr (pn_line)
         select case (char (parse_node_get_rule_key (pn_code1)))
         case ("integer")
```

2721

```
            if (parse_node_get_integer (pn_code1) == code1) then
               pn_code2 => parse_node_get_next_ptr (pn_code1)
               if (associated (pn_code2)) then
                  select case (char (parse_node_get_rule_key (pn_code2)))
                  case ("integer")
                     if (parse_node_get_integer (pn_code2) == code2) then
                        pn_item => parse_node_get_next_ptr (pn_code2)
                        return
                     end if
                  end select
               end if
            end if
         end select
         pn_data => parse_node_get_next_ptr (pn_data)
      end do
      pn_item => null ()
   end subroutine slha_next_index_pair_ptr
```

Handle info data

Return all strings with index `i`. The result is an allocated string array. Since
we do not know the number of matching entries in advance, we build an inter-
mediate list which is transferred to the final array and deleted before exiting.

⟨*SLHA: procedures*⟩+≡

```
   subroutine retrieve_strings_in_block (pn_block, code, str_array)
      type(parse_node_t), intent(in), target :: pn_block
      integer, intent(in) :: code
      type(string_t), dimension(:), allocatable, intent(out) :: str_array
      type(parse_node_t), pointer :: pn_data, pn_item
      type :: str_entry_t
         type(string_t) :: str
         type(str_entry_t), pointer :: next => null ()
      end type str_entry_t
      type(str_entry_t), pointer :: first => null ()
      type(str_entry_t), pointer :: current => null ()
      integer :: n
      n = 0
      call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
      if (associated (pn_item)) then
         n = n + 1
         allocate (first)
         first%str = parse_node_get_string (pn_item)
         current => first
         do while (associated (pn_data))
            pn_data => parse_node_get_next_ptr (pn_data)
            call slha_next_index_ptr (pn_data, pn_item, code)
            if (associated (pn_item)) then
               n = n + 1
               allocate (current%next)
               current => current%next
               current%str = parse_node_get_string (pn_item)
            end if
```

```
         end do
         allocate (str_array (n))
         n = 0
         do while (associated (first))
            n = n + 1
            current => first
            str_array(n) = current%str
            first => first%next
            deallocate (current)
         end do
      else
         allocate (str_array (0))
      end if
   end subroutine retrieve_strings_in_block
```

Transfer data from SLHA to variables

Extract real parameter with index i. If it does not exist, retrieve it from the variable list, using the given name.

⟨*SLHA: procedures*⟩+≡
```
   function get_parameter_in_block (pn_block, code, name, var_list) result (var)
      real(default) :: var
      type(parse_node_t), intent(in), target :: pn_block
      integer, intent(in) :: code
      type(string_t), intent(in) :: name
      type(var_list_t), intent(in), target :: var_list
      type(parse_node_t), pointer :: pn_data, pn_item
      call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
      if (associated (pn_item)) then
         var = get_real_parameter (pn_item)
      else
         var = var_list_get_rval (var_list, name)
      end if
   end function get_parameter_in_block
```

Extract a real data item with index i. If it does exist, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

⟨*SLHA: procedures*⟩+≡
```
   subroutine set_data_item (pn_block, code, name, var_list)
      type(parse_node_t), intent(in), target :: pn_block
      integer, intent(in) :: code
      type(string_t), intent(in) :: name
      type(var_list_t), intent(inout), target :: var_list
      type(parse_node_t), pointer :: pn_data, pn_item
      call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
      if (associated (pn_item)) then
         call var_list_set_real &
              (var_list, name,  get_real_parameter (pn_item), &
               is_known=.true., ignore=.true.)
      end if
   end subroutine set_data_item
```

Extract a real matrix element with index `i,j`. If it does exists, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

⟨*SLHA: procedures*⟩+≡
```
subroutine set_matrix_element (pn_block, code1, code2, name, var_list)
  type(parse_node_t), intent(in), target :: pn_block
  integer, intent(in) :: code1, code2
  type(string_t), intent(in) :: name
  type(var_list_t), intent(inout), target :: var_list
  type(parse_node_t), pointer :: pn_data, pn_item
  call slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
  if (associated (pn_item)) then
     call var_list_set_real &
          (var_list, name,  get_real_parameter (pn_item), &
           is_known=.true., ignore=.true.)
  end if
end subroutine set_matrix_element
```

Transfer data from variables to SLHA

Get a real/integer parameter with index `i` from the variable list and write it to the current output file. In the integer case, we account for the fact that the variable is type real. If it does not exist, do nothing.

⟨*SLHA: procedures*⟩+≡
```
subroutine write_integer_data_item (u, code, name, var_list, comment)
  integer, intent(in) :: u
  integer, intent(in) :: code
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in) :: var_list
  character(*), intent(in) :: comment
  integer :: item
  if (var_list_exists (var_list, name)) then
     item = nint (var_list_get_rval (var_list, name))
     call write_integer_parameter (u, code, item, comment)
  end if
end subroutine write_integer_data_item

subroutine write_real_data_item (u, code, name, var_list, comment)
  integer, intent(in) :: u
  integer, intent(in) :: code
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in) :: var_list
  character(*), intent(in) :: comment
  real(default) :: item
  if (var_list_exists (var_list, name)) then
     item = var_list_get_rval (var_list, name)
     call write_real_parameter (u, code, item, comment)
  end if
end subroutine write_real_data_item
```

Get a real data item with two integer indices from the variable list and write it to the current output file. If it does not exist, do nothing.

⟨*SLHA: procedures*⟩+≡
```
subroutine write_matrix_element (u, code1, code2, name, var_list, comment)
  integer, intent(in) :: u
  integer, intent(in) :: code1, code2
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in) :: var_list
  character(*), intent(in) :: comment
  real(default) :: item
  if (var_list_exists (var_list, name)) then
     item = var_list_get_rval (var_list, name)
     call write_real_matrix_element (u, code1, code2, item, comment)
  end if
end subroutine write_matrix_element
```

20.0.8 Auxiliary function

Write a block header.

⟨*SLHA: procedures*⟩+≡
```
subroutine write_block_header (u, name, comment)
  integer, intent(in) :: u
  character(*), intent(in) :: name, comment
  write (u, "(A,1x,A,3x,'#',1x,A)")  "BLOCK", name, comment
end subroutine write_block_header
```

Extract a real parameter that may be defined real or integer, signed or unsigned.

⟨*SLHA: procedures*⟩+≡
```
function get_real_parameter (pn_item) result (var)
  real(default) :: var
  type(parse_node_t), intent(in), target :: pn_item
  type(parse_node_t), pointer :: pn_sign, pn_var
  integer :: sign
  select case (char (parse_node_get_rule_key (pn_item)))
  case ("signed_number")
     pn_sign => parse_node_get_sub_ptr (pn_item)
     pn_var  => parse_node_get_next_ptr (pn_sign)
     select case (char (parse_node_get_key (pn_sign)))
     case ("+");  sign = +1
     case ("-");  sign = -1
     end select
  case default
     sign = +1
     pn_var => pn_item
  end select
  select case (char (parse_node_get_rule_key (pn_var)))
  case ("integer");  var = sign * parse_node_get_integer (pn_var)
  case ("real");     var = sign * parse_node_get_real (pn_var)
  end select
end function get_real_parameter
```

2725

Auxiliary: Extract an integer parameter that may be defined signed or unsigned.
A real value is an error.

⟨*SLHA: procedures*⟩+≡

```
    function get_integer_parameter (pn_item) result (var)
      integer :: var
      type(parse_node_t), intent(in), target :: pn_item
      type(parse_node_t), pointer :: pn_sign, pn_var
      integer :: sign
      select case (char (parse_node_get_rule_key (pn_item)))
      case ("signed_integer")
         pn_sign => parse_node_get_sub_ptr (pn_item)
         pn_var  => parse_node_get_next_ptr (pn_sign)
         select case (char (parse_node_get_key (pn_sign)))
         case ("+");  sign = +1
         case ("-");  sign = -1
         end select
      case ("integer")
         sign = +1
         pn_var => pn_item
      case default
         call parse_node_write (pn_var)
         call msg_error ("SLHA: Integer parameter expected")
         var = 0
         return
      end select
      var = sign * parse_node_get_integer (pn_var)
    end function get_integer_parameter
```

Write an integer parameter with a single index directly to file, using the required
output format.

⟨*SLHA: procedures*⟩+≡

```
    subroutine write_integer_parameter (u, code, item, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code
      integer, intent(in) :: item
      character(*), intent(in) :: comment
  1   format (1x, I9, 3x, 3x, I9, 4x, 3x, '#', 1x, A)
      write (u, 1)  code, item, comment
    end subroutine write_integer_parameter
```

Write a real parameter with two indices directly to file, using the required output
format.

⟨*SLHA: procedures*⟩+≡

```
    subroutine write_real_parameter (u, code, item, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code
      real(default), intent(in) :: item
      character(*), intent(in) :: comment
  1   format (1x, I9, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
      write (u, 1)  code, item, comment
    end subroutine write_real_parameter
```

Write a real parameter with a single index directly to file, using the required output format.

⟨*SLHA: procedures*⟩+≡
```
    subroutine write_real_matrix_element (u, code1, code2, item, comment)
      integer, intent(in) :: u
      integer, intent(in) :: code1, code2
      real(default), intent(in) :: item
      character(*), intent(in) :: comment
  1   format (1x, I2, 1x, I2, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
      write (u, 1)  code1, code2, item, comment
    end subroutine write_real_matrix_element
```

The concrete SLHA interpreter

SLHA codes for particular physics models

⟨*SLHA: parameters*⟩+≡
```
    integer, parameter :: MDL_MSSM = 0
    integer, parameter :: MDL_NMSSM = 1
```

Take the parse tree and extract relevant data. Select the correct model and store all data that is present in the appropriate variable list. Finally, update the variable record.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_interpret_parse_tree &
        (parse_tree, os_data, model_list, model, input, spectrum, decays)
      type(parse_tree_t), intent(in) :: parse_tree
      type(os_data_t), intent(in) :: os_data
      type(model_list_t), intent(inout) :: model_list
      type(model_t), pointer, intent(inout) :: model
      logical, intent(in) :: input, spectrum, decays
      logical :: errors
      integer :: mssm_type
      call slha_handle_MODSEL (parse_tree, os_data, model_list, model, mssm_type)
      if (associated (model)) then
         if (input) then
            call slha_handle_SMINPUTS (parse_tree, model)
            call slha_handle_MINPAR (parse_tree, model, mssm_type)
         end if
         if (spectrum) then
            call slha_handle_info_block (parse_tree, "SPINFO", errors)
            if (errors)  return
            call slha_handle_MASS (parse_tree, model)
            call slha_handle_matrix_block (parse_tree, "NMIX", "mn_", 4, 4, model)
            call slha_handle_matrix_block (parse_tree, "NMNMIX", "mixn_", 5, 5, model)
            call slha_handle_matrix_block (parse_tree, "UMIX", "mu_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "VMIX", "mv_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "STOPMIX", "mt_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "SBOTMIX", "mb_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "STAUMIX", "ml_", 2, 2, model)
            call slha_handle_matrix_block (parse_tree, "NMHMIX", "mixh0_", 3, 3, model)
            call slha_handle_matrix_block (parse_tree, "NMAMIX", "mixa0_", 2, 3, model)
            call slha_handle_ALPHA (parse_tree, model)
```

```
        call slha_handle_HMIX (parse_tree, model)
        call slha_handle_NMSSMRUN (parse_tree, model)
        call slha_handle_matrix_block (parse_tree, "AU", "Au_", 3, 3, model)
        call slha_handle_matrix_block (parse_tree, "AD", "Ad_", 3, 3, model)
        call slha_handle_matrix_block (parse_tree, "AE", "Ae_", 3, 3, model)
      end if
      if (decays) then
        call slha_handle_info_block (parse_tree, "DCINFO", errors)
        if (errors)  return
        call slha_handle_decays (parse_tree, model)
      end if
    end if
  end if
end subroutine slha_interpret_parse_tree
```

Info blocks

Handle the informational blocks SPINFO and DCINFO. The first two items are
program name and version. Items with index 3 are warnings. Items with index
4 are errors. We reproduce these as WHIZARD warnings and errors.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_handle_info_block (parse_tree, block_name, errors)
  type(parse_tree_t), intent(in) :: parse_tree
  character(*), intent(in) :: block_name
  logical, intent(out) :: errors
  type(parse_node_t), pointer :: pn_block
  type(string_t), dimension(:), allocatable :: msg
  integer :: i
  pn_block => slha_get_block_ptr &
       (parse_tree, var_str (block_name), required=.true.)
  if (.not. associated (pn_block)) then
    call msg_error ("SLHA: Missing info block '" &
         // trim (block_name) // "'; ignored.")
    errors = .true.
    return
  end if
  select case (block_name)
  case ("SPINFO")
    call msg_message ("SLHA: SUSY spectrum program info:")
  case ("DCINFO")
    call msg_message ("SLHA: SUSY decay program info:")
  end select
  call retrieve_strings_in_block (pn_block, 1, msg)
  do i = 1, size (msg)
    call msg_message ("SLHA: " // char (msg(i)))
  end do
  call retrieve_strings_in_block (pn_block, 2, msg)
  do i = 1, size (msg)
    call msg_message ("SLHA: " // char (msg(i)))
  end do
  call retrieve_strings_in_block (pn_block, 3, msg)
  do i = 1, size (msg)
    call msg_warning ("SLHA: " // char (msg(i)))
```

```
      end do
      call retrieve_strings_in_block (pn_block, 4, msg)
      do i = 1, size (msg)
          call msg_error ("SLHA: " // char (msg(i)))
      end do
      errors = size (msg) > 0
   end subroutine slha_handle_info_block
```

MODSEL

Handle the overall model definition. Only certain models are recognized. The soft-breaking model templates that determine the set of input parameters:

⟨*SLHA: parameters*⟩+≡
```
    integer, parameter :: MSSM_GENERIC = 0
    integer, parameter :: MSSM_SUGRA = 1
    integer, parameter :: MSSM_GMSB = 2
    integer, parameter :: MSSM_AMSB = 3
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_handle_MODSEL &
        (parse_tree, os_data, model_list, model, mssm_type)
      type(parse_tree_t), intent(in) :: parse_tree
      type(os_data_t), intent(in) :: os_data
      type(model_list_t), intent(inout) :: model_list
      type(model_t), pointer, intent(inout) :: model
      integer, intent(out) :: mssm_type
      type(parse_node_t), pointer :: pn_block, pn_data, pn_item
      type(string_t) :: model_name
      type(string_t) :: filename
      pn_block => slha_get_block_ptr &
          (parse_tree, var_str ("MODSEL"), required=.true.)
      call slha_find_index_ptr (pn_block, pn_data, pn_item, 1)
      if (associated (pn_item)) then
         mssm_type = get_integer_parameter (pn_item)
      else
         mssm_type = MSSM_GENERIC
      end if
      call slha_find_index_ptr (pn_block, pn_data, pn_item, 3)
      if (associated (pn_item)) then
         select case (parse_node_get_integer (pn_item))
         case (MDL_MSSM);  model_name = "MSSM"
         case (MDL_NMSSM); model_name = "NMSSM"
         case default
            call msg_fatal ("SLHA: unknown model code in MODSEL")
            return
         end select
      else
         model_name = "MSSM"
      end if
      call slha_find_index_ptr (pn_block, pn_data, pn_item, 4)
      if (associated (pn_item)) then
        call msg_fatal (" R-parity violation is currently not supported by WHIZARD.")
```

2729

```
      end if
      call slha_find_index_ptr (pn_block, pn_data, pn_item, 5)
      if (associated (pn_item)) then
        call msg_fatal (" CP violation is currently not supported by WHIZARD.")
      end if
      select case (char (model_name))
        case ("MSSM")
           select case (char (model%get_name ()))
             case ("MSSM","MSSM_CKM","MSSM_Grav","MSSM_Hgg")
                model_name = model%get_name ()
             case default
               call msg_fatal (" User-defined model and model in SLHA input file do not match.")
           end select
        case ("NMSSM")
           select case (char (model%get_name ()))
             case ("NMSSM","NMSSM_CKM","NMSSM_Hgg")
                model_name = model%get_name ()
             case default
               call msg_fatal (" User-defined model and model in SLHA input file do not match.")
           end select
        case default
           call msg_fatal (" SLHA model selection failure.")
      end select
      filename = model_name // ".mdl"
      model => null ()
      call model_list%read_model (model_name, filename, os_data, model)
      if (associated (model)) then
         call msg_message ("SLHA: Initializing model '" &
              // char (model_name) // "'")
      else
         call msg_fatal ("SLHA: Initialization failed for model '" &
              // char (model_name) // "'")
      end if
    end subroutine slha_handle_MODSEL
```

Write a MODSEL block, based on the contents of the current model.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_write_MODSEL (u, model, mssm_type)
      integer, intent(in) :: u
      type(model_t), intent(in), target :: model
      integer, intent(out) :: mssm_type
      type(var_list_t), pointer :: var_list
      integer :: model_id
      type(string_t) :: mtype_string
      var_list => model_get_var_list_ptr (model)
      if (var_list_exists (var_list, var_str ("mtype"))) then
         mssm_type = nint (var_list_get_rval (var_list, var_str ("mtype")))
      else
         call msg_error ("SLHA: parameter 'mtype' (SUSY breaking scheme) " &
              // "is unknown in current model, no SLHA output possible")
         mssm_type = -1
         return
      end if
```

```
      call write_block_header (u, "MODSEL", "SUSY model selection")
      select case (mssm_type)
      case (0);  mtype_string = "Generic MSSM"
      case (1);  mtype_string = "SUGRA"
      case (2);  mtype_string = "GMSB"
      case (3);  mtype_string = "AMSB"
      case default
         mtype_string = "unknown"
      end select
      call write_integer_parameter (u, 1, mssm_type, &
           "SUSY-breaking scheme: " // char (mtype_string))
      select case (char (model%get_name ()))
      case ("MSSM");  model_id = MDL_MSSM
      case ("NMSSM"); model_id = MDL_NMSSM
      case default
         model_id = 0
      end select
      call write_integer_parameter (u, 3, model_id, &
           "SUSY model type: " // char (model%get_name ()))
    end subroutine slha_write_MODSEL
```

SMINPUTS

Read SM parameters and update the variable list accordingly. If a parameter is
not defined in the block, we use the previous value from the model variable list.
For the basic parameters we have to do a small recalculation, since SLHA uses
the G_F-α-m_Z scheme, while WHIZARD derives them from G_F, m_W, and m_Z.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_handle_SMINPUTS (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block
      real(default) :: alpha_em_i, GF, alphas, mZ
      real(default) :: ee, vv, cw_sw, cw2, mW
      real(default) :: mb, mtop, mtau
      type(var_list_t), pointer :: var_list
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("SMINPUTS"), required=.true.)
      if (.not. (associated (pn_block)))  return
      alpha_em_i = &
           get_parameter_in_block (pn_block, 1, var_str ("alpha_em_i"), var_list)
      GF = get_parameter_in_block (pn_block, 2, var_str ("GF"), var_list)
      alphas = &
           get_parameter_in_block (pn_block, 3, var_str ("alphas"), var_list)
      mZ   = get_parameter_in_block (pn_block, 4, var_str ("mZ"), var_list)
      mb   = get_parameter_in_block (pn_block, 5, var_str ("mb"), var_list)
      mtop = get_parameter_in_block (pn_block, 6, var_str ("mtop"), var_list)
      mtau = get_parameter_in_block (pn_block, 7, var_str ("mtau"), var_list)
      ee = sqrt (4 * pi / alpha_em_i)
      vv = 1 / sqrt (sqrt (2._default) * GF)
      cw_sw = ee * vv / (2 * mZ)
```

2731

```
if (2*cw_sw <= 1) then
   cw2 = (1 + sqrt (1 - 4 * cw_sw**2)) / 2
   mW = mZ * sqrt (cw2)
   call var_list_set_real (var_list, var_str ("GF"), GF, .true.)
   call var_list_set_real (var_list, var_str ("mZ"), mZ, .true.)
   call var_list_set_real (var_list, var_str ("mW"), mW, .true.)
   call var_list_set_real (var_list, var_str ("mtau"), mtau, .true.)
   call var_list_set_real (var_list, var_str ("mb"), mb, .true.)
   call var_list_set_real (var_list, var_str ("mtop"), mtop, .true.)
   call var_list_set_real (var_list, var_str ("alphas"), alphas, .true.)
else
   call msg_fatal ("SLHA: Unphysical SM parameter values")
   return
end if
end subroutine slha_handle_SMINPUTS
```

Write a SMINPUTS block.

⟨*SLHA: procedures*⟩+≡
```
subroutine slha_write_SMINPUTS (u, model)
  integer, intent(in) :: u
  type(model_t), intent(in), target :: model
  type(var_list_t), pointer :: var_list
  var_list => model_get_var_list_ptr (model)
  call write_block_header (u, "SMINPUTS", "SM input parameters")
  call write_real_data_item (u, 1, var_str ("alpha_em_i"), var_list, &
       "Inverse electromagnetic coupling alpha (Z pole)")
  call write_real_data_item (u, 2, var_str ("GF"), var_list, &
       "Fermi constant")
  call write_real_data_item (u, 3, var_str ("alphas"), var_list, &
       "Strong coupling alpha_s (Z pole)")
  call write_real_data_item (u, 4, var_str ("mZ"), var_list, &
       "Z mass")
  call write_real_data_item (u, 5, var_str ("mb"), var_list, &
       "b running mass (at mb)")
  call write_real_data_item (u, 6, var_str ("mtop"), var_list, &
       "top mass")
  call write_real_data_item (u, 7, var_str ("mtau"), var_list, &
       "tau mass")
end subroutine slha_write_SMINPUTS
```

MINPAR

The block of SUSY input parameters. They are accessible to WHIZARD, but
they only get used when an external spectrum generator is invoked. The precise
set of parameters depends on the type of SUSY breaking, which by itself is one
of the parameters.

⟨*SLHA: procedures*⟩+≡
```
subroutine slha_handle_MINPAR (parse_tree, model, mssm_type)
  type(parse_tree_t), intent(in) :: parse_tree
  type(model_t), intent(inout), target :: model
  integer, intent(in) :: mssm_type
  type(var_list_t), pointer :: var_list
```

```
      type(parse_node_t), pointer :: pn_block
      var_list => model_get_var_list_ptr (model)
      call var_list_set_real (var_list, &
           var_str ("mtype"),  real(mssm_type, default), is_known=.true.)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("MINPAR"), required=.true.)
      select case (mssm_type)
      case (MSSM_SUGRA)
         call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
         call set_data_item (pn_block, 2, var_str ("m_half"), var_list)
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
         call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
         call set_data_item (pn_block, 5, var_str ("A0"), var_list)
      case (MSSM_GMSB)
         call set_data_item (pn_block, 1, var_str ("Lambda"), var_list)
         call set_data_item (pn_block, 2, var_str ("M_mes"), var_list)
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
         call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
         call set_data_item (pn_block, 5, var_str ("N_5"), var_list)
         call set_data_item (pn_block, 6, var_str ("c_grav"), var_list)
      case (MSSM_AMSB)
         call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
         call set_data_item (pn_block, 2, var_str ("m_grav"), var_list)
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
         call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
      case default
         call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
      end select
    end subroutine slha_handle_MINPAR
```

Write a MINPAR block as appropriate for the current model type.
⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_write_MINPAR (u, model, mssm_type)
      integer, intent(in) :: u
      type(model_t), intent(in), target :: model
      integer, intent(in) :: mssm_type
      type(var_list_t), pointer :: var_list
      var_list => model_get_var_list_ptr (model)
      call write_block_header (u, "MINPAR", "Basic SUSY input parameters")
      select case (mssm_type)
      case (MSSM_SUGRA)
         call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
              "Common scalar mass")
         call write_real_data_item (u, 2, var_str ("m_half"), var_list, &
              "Common gaugino mass")
         call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
              "tan(beta)")
         call write_integer_data_item (u, 4, &
              var_str ("sgn_mu"), var_list, &
              "Sign of mu")
         call write_real_data_item (u, 5, var_str ("A0"), var_list, &
              "Common trilinear coupling")
      case (MSSM_GMSB)
         call write_real_data_item (u, 1, var_str ("Lambda"), var_list, &
```

```
            "Soft-breaking scale")
         call write_real_data_item (u, 2, var_str ("M_mes"), var_list, &
            "Messenger scale")
         call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
            "tan(beta)")
         call write_integer_data_item (u, 4, &
            var_str ("sgn_mu"), var_list, &
            "Sign of mu")
         call write_integer_data_item (u, 5, var_str ("N_5"), var_list, &
            "Messenger index")
         call write_real_data_item (u, 6, var_str ("c_grav"), var_list, &
            "Gravitino mass factor")
      case (MSSM_AMSB)
         call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
            "Common scalar mass")
         call write_real_data_item (u, 2, var_str ("m_grav"), var_list, &
            "Gravitino mass")
         call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
            "tan(beta)")
         call write_integer_data_item (u, 4, &
            var_str ("sgn_mu"), var_list, &
            "Sign of mu")
      case default
         call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
            "tan(beta)")
      end select
    end subroutine slha_write_MINPAR
```

Mass spectrum

Set masses. Since the particles are identified by PDG code, read the line and try
to set the appropriate particle mass in the current model. At the end, update
parameters, just in case the W or Z mass was included.

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_handle_MASS (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block, pn_data, pn_line, pn_code
      type(parse_node_t), pointer :: pn_mass
      integer :: pdg
      real(default) :: mass
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("MASS"), required=.true.)
      if (.not. (associated (pn_block)))  return
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      do while (associated (pn_data))
         pn_line => parse_node_get_sub_ptr (pn_data, 2)
         pn_code => parse_node_get_sub_ptr (pn_line)
         if (associated (pn_code)) then
            pdg = get_integer_parameter (pn_code)
            pn_mass => parse_node_get_next_ptr (pn_code)
            if (associated (pn_mass)) then
```

2734

```
            mass = get_real_parameter (pn_mass)
            call model_set_particle_mass (model, pdg, mass)
         else
            call msg_error ("SLHA: Block MASS: Missing mass value")
         end if
      else
         call msg_error ("SLHA: Block MASS: Missing PDG code")
      end if
      pn_data => parse_node_get_next_ptr (pn_data)
   end do
end subroutine slha_handle_MASS
```

Widths

Set widths. For each DECAY block, extract the header, read the PDG code
and width, and try to set the appropriate particle width in the current model.

⟨*SLHA: procedures*⟩+≡
```
   subroutine slha_handle_decays (parse_tree, model)
     type(parse_tree_t), intent(in) :: parse_tree
     type(model_t), intent(inout), target :: model
     type(parse_node_t), pointer :: pn_decay, pn_decay_spec, pn_code, pn_width
     integer :: pdg
     real(default) :: width
     pn_decay => slha_get_first_decay_ptr (parse_tree)
     do while (associated (pn_decay))
        pn_decay_spec => parse_node_get_sub_ptr (pn_decay, 2)
        pn_code => parse_node_get_sub_ptr (pn_decay_spec)
        pdg = get_integer_parameter (pn_code)
        pn_width => parse_node_get_next_ptr (pn_code)
        width = get_real_parameter (pn_width)
        call model_set_particle_width (model, pdg, width)
        pn_decay => slha_get_next_decay_ptr (pn_decay)
     end do
   end subroutine slha_handle_decays
```

Mixing matrices

Read mixing matrices. We can treat all matrices by a single procedure if we
just know the block name, variable prefix, and matrix dimension. The matrix
dimension must be less than 10. For the pseudoscalar Higgses in NMSSM-type
models we need off-diagonal matrices, so we generalize the definition.

⟨*SLHA: procedures*⟩+≡
```
   subroutine slha_handle_matrix_block &
        (parse_tree, block_name, var_prefix, dim1, dim2, model)
     type(parse_tree_t), intent(in) :: parse_tree
     character(*), intent(in) :: block_name, var_prefix
     integer, intent(in) :: dim1, dim2
     type(model_t), intent(inout), target :: model
     type(parse_node_t), pointer :: pn_block
     type(var_list_t), pointer :: var_list
     integer :: i, j
```

```
      character(len=len(var_prefix)+2) :: var_name
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str (block_name), required=.false.)
      if (.not. (associated (pn_block)))  return
      do i = 1, dim1
         do j = 1, dim2
            write (var_name, "(A,I1,I1)")  var_prefix, i, j
            call set_matrix_element (pn_block, i, j, var_str (var_name), var_list)
         end do
      end do
    end subroutine slha_handle_matrix_block
```

Higgs data

Read the block ALPHA which holds just the Higgs mixing angle.

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_handle_ALPHA (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block, pn_line, pn_data, pn_item
      type(var_list_t), pointer :: var_list
      real(default) :: al_h
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("ALPHA"), required=.false.)
      if (.not. (associated (pn_block)))  return
      pn_data => parse_node_get_sub_ptr (pn_block, 4)
      pn_line => parse_node_get_sub_ptr (pn_data, 2)
      pn_item => parse_node_get_sub_ptr (pn_line)
      if (associated (pn_item)) then
         al_h = get_real_parameter (pn_item)
         call var_list_set_real (var_list, var_str ("al_h"), al_h, &
              is_known=.true., ignore=.true.)
      end if
    end subroutine slha_handle_ALPHA
```

Read the block HMIX for the Higgs mixing parameters

⟨*SLHA: procedures*⟩+≡

```
    subroutine slha_handle_HMIX (parse_tree, model)
      type(parse_tree_t), intent(in) :: parse_tree
      type(model_t), intent(inout), target :: model
      type(parse_node_t), pointer :: pn_block
      type(var_list_t), pointer :: var_list
      var_list => model_get_var_list_ptr (model)
      pn_block => slha_get_block_ptr &
           (parse_tree, var_str ("HMIX"), required=.false.)
      if (.not. (associated (pn_block)))  return
      call set_data_item (pn_block, 1, var_str ("mu_h"), var_list)
      call set_data_item (pn_block, 2, var_str ("tanb_h"), var_list)
    end subroutine slha_handle_HMIX
```

Read the block NMSSMRUN for the specific NMSSM parameters

⟨SLHA: procedures⟩+≡
```
subroutine slha_handle_NMSSMRUN (parse_tree, model)
  type(parse_tree_t), intent(in) :: parse_tree
  type(model_t), intent(inout), target :: model
  type(parse_node_t), pointer :: pn_block
  type(var_list_t), pointer :: var_list
  var_list => model_get_var_list_ptr (model)
  pn_block => slha_get_block_ptr &
       (parse_tree, var_str ("NMSSMRUN"), required=.false.)
  if (.not. (associated (pn_block)))  return
  call set_data_item (pn_block, 1, var_str ("ls"), var_list)
  call set_data_item (pn_block, 2, var_str ("ks"), var_list)
  call set_data_item (pn_block, 3, var_str ("a_ls"), var_list)
  call set_data_item (pn_block, 4, var_str ("a_ks"), var_list)
  call set_data_item (pn_block, 5, var_str ("nmu"), var_list)
  end subroutine slha_handle_NMSSMRUN
```

20.0.9 Parser

Read a SLHA file from stream, including preprocessing, and make up a parse tree.

⟨SLHA: procedures⟩+≡
```
subroutine slha_parse_stream (stream, parse_tree)
  type(stream_t), intent(inout), target :: stream
  type(parse_tree_t), intent(out) :: parse_tree
  type(ifile_t) :: ifile
  type(lexer_t) :: lexer
  type(stream_t), target :: stream_tmp
  call slha_preprocess (stream, ifile)
  call stream_init (stream_tmp, ifile)
  call lexer_init_slha (lexer)
  call lexer_assign_stream (lexer, stream_tmp)
  call parse_tree_init (parse_tree, syntax_slha, lexer)
  call lexer_final (lexer)
  call stream_final (stream_tmp)
  call ifile_final (ifile)
end subroutine slha_parse_stream
```

Read a SLHA file chosen by name. Check first the current directory, then the directory where SUSY input files should be located.

⟨SLHA: procedures⟩+≡
```
subroutine slha_parse_file (file, os_data, parse_tree)
  type(string_t), intent(in) :: file
  type(os_data_t), intent(in) :: os_data
  type(parse_tree_t), intent(out) :: parse_tree
  logical :: exist
  type(string_t) :: filename
  type(stream_t), target :: stream
  call msg_message ("Reading SLHA input file '" // char (file) // "'")
  filename = file
```

```
      inquire (file=char(filename), exist=exist)
      if (.not. exist) then
         filename = os_data%whizard_susypath // "/" // file
         inquire (file=char(filename), exist=exist)
         if (.not. exist) then
            call msg_fatal ("SLHA input file '" // char (file) // "' not found")
            return
         end if
      end if
      call stream_init (stream, char (filename))
      call slha_parse_stream (stream, parse_tree)
      call stream_final (stream)
   end subroutine slha_parse_file
```

20.0.10   API

Read the SLHA file, parse it, and interpret the parse tree. The model parameters
retrieved from the file will be inserted into the appropriate model, which is
loaded and modified in the background. The pointer to this model is returned
as the last argument.

⟨*SLHA: public*⟩+≡
```
   public :: slha_read_file
```

⟨*SLHA: procedures*⟩+≡
```
   subroutine slha_read_file &
        (file, os_data, model_list, model, input, spectrum, decays)
     type(string_t), intent(in) :: file
     type(os_data_t), intent(in) :: os_data
     type(model_list_t), intent(inout) :: model_list
     type(model_t), pointer, intent(inout) :: model
     logical, intent(in) :: input, spectrum, decays
     type(parse_tree_t) :: parse_tree
     call slha_parse_file (file, os_data, parse_tree)
     if (associated (parse_tree_get_root_ptr (parse_tree))) then
        call slha_interpret_parse_tree &
             (parse_tree, os_data, model_list, model, input, spectrum, decays)
        call parse_tree_final (parse_tree)
        call model_parameters_update (model)
     end if
   end subroutine slha_read_file
```

Write the SLHA contents, as far as possible, to external file.

⟨*SLHA: public*⟩+≡
```
   public :: slha_write_file
```

⟨*SLHA: procedures*⟩+≡
```
   subroutine slha_write_file (file, model, input, spectrum, decays)
     type(string_t), intent(in) :: file
     type(model_t), target, intent(in) :: model
     logical, intent(in) :: input, spectrum, decays
     integer :: mssm_type
     integer :: u
```

```
    u = free_unit ()
    call msg_message ("Writing SLHA output file '" // char (file) // "'")
    open (unit=u, file=char(file), action="write", status="replace")
    write (u, "(A)")  "# SUSY Les Houches Accord"
    write (u, "(A)")  "# Output generated by " // trim (VERSION_STRING)
    call slha_write_MODSEL (u, model, mssm_type)
    if (input) then
       call slha_write_SMINPUTS (u, model)
       call slha_write_MINPAR (u, model, mssm_type)
    end if
    if (spectrum) then
       call msg_bug ("SLHA: spectrum output not supported yet")
    end if
    if (decays) then
       call msg_bug ("SLHA: decays output not supported yet")
    end if
    close (u)
  end subroutine slha_write_file
```

20.0.11   Unit tests

⟨*SLHA: public*⟩+≡
```
    public :: slha_test
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine slha_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SLHA: execute tests⟩
    end subroutine slha_test
```

Checking the basics of the SLHA interface.

⟨*SLHA: execute tests*⟩≡
```
    call test (slha_1, "slha_1", &
         "check SLHA interface", &
         u, results)
```

⟨*SLHA: tests*⟩≡
```
    subroutine slha_1 (u)
      integer, intent(in) :: u
      type(os_data_t), pointer :: os_data => null ()
      type(parse_tree_t), pointer :: parse_tree => null ()
      integer :: u_file, iostat
      character(80) :: buffer
      character(*), parameter :: file_slha = "slha_test.dat"
      type(model_list_t) :: model_list
      type(model_t), pointer :: model => null ()

      write (u, "(A)")  "* Test output: SLHA Interface"
      write (u, "(A)")  "*   Purpose: test SLHA file reading and writing"
      write (u, "(A)")

      write (u, "(A)")  "* Initializing"
```

```
write (u, "(A)")

allocate (os_data)
allocate (parse_tree)
call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model &
     (var_str("MSSM"), var_str("MSSM.mdl"), os_data, model)
call syntax_slha_init ()

write (u, "(A)")  "* Reading SLHA file sps1ap_decays.slha"
write (u, "(A)")

call slha_parse_file (var_str ("sps1ap_decays.slha"), os_data, parse_tree)

write (u, "(A)")  "* Writing the parse tree:"
write (u, "(A)")

call parse_tree_write (parse_tree, u)

write (u, "(A)")  "* Interpreting the parse tree"
write (u, "(A)")

call slha_interpret_parse_tree (parse_tree, os_data, model_list, model, &
     input=.true., spectrum=.true., decays=.true.)
call parse_tree_final (parse_tree)

write (u, "(A)")  "* Writing out the list of variables (reals only):"
write (u, "(A)")

call var_list_write (model_get_var_list_ptr (model), &
     only_type = V_REAL, unit = u)

write (u, "(A)")
write (u, "(A)")  "* Writing SLHA output to '" // file_slha // "'"
write (u, "(A)")

call slha_write_file (var_str (file_slha), model, input=.true., &
     spectrum=.false., decays=.false.)
u_file = free_unit ()
open (u_file, file = file_slha, action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:37) == "# Output generated by WHIZARD version") then
      buffer = "[...]"
   end if
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
write (u, "(A)")
```

```
    call parse_tree_final (parse_tree)
    deallocate (parse_tree)
    deallocate (os_data)

    write (u, "(A)")  "* Test output end: slha_1"
    write (u, "(A)")

end subroutine slha_1
```

Chapter 21

NLO infrastructure

21.1 BLHA interface

These modules implement the communication with one loop matrix element providers according to the Binoth LesHouches Accord Interface. The actual matrix element(s) are loaded as a dynamic library.

The module is split into a configuration interface which manages configuration and handles the request and contract files, a module which interfaces the OLP matrix elements and a driver.

⟨blha_config.f90⟩≡
  ⟨*File header*⟩

  module blha_config

  ⟨*Use strings*⟩
    use constants !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use md5
    use models
    use flavors
    use quantum_numbers
    use pdg_arrays
    use sorting
    use lexers
    use parser
    use syntax_rules
    use ifiles
    use limits, only: EOF !NODEP!

  ⟨*Standard module head*⟩

  ⟨*BLHA config: public*⟩

  ⟨*BLHA config: parameters*⟩

  ⟨*BLHA config: types*⟩

⟨*BLHA config: variables*⟩

⟨*BLHA config: interfaces*⟩

```
contains
```

⟨*BLHA config: procedures*⟩

```
end module blha_config
```

⟨`blha_interface.f90`⟩≡
  ⟨*File header*⟩

```
module blha_interface
```

⟨*Use strings*⟩
```
  use constants !NODEP!
```
⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use sm_physics !NODEP!
  use md5
  use lorentz !NODEP!
  use models
  use flavors
  use quantum_numbers
  use interactions
  use evaluators
  use particles
  use quantum_numbers
  use blha_config
  use, intrinsic :: iso_c_binding !NODEP!
  use os_interface
```

⟨*Standard module head*⟩

⟨*BLHA interface: public*⟩

⟨*BLHA interface: types*⟩

⟨*BLHA interface: interfaces*⟩

```
contains
```

⟨*BLHA interface: procedures*⟩

```
end module blha_interface
```
⟨`blha_driver.f90`⟩≡
  ⟨*File header*⟩

```
module blha_driver
```

⟨*Use strings*⟩
```
  use constants !NODEP!
```

⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use sm_physics !NODEP!
  use md5
  use lorentz !NODEP!
  use models
  use flavors
  use quantum_numbers
  use interactions
  use evaluators
  use particles
  use quantum_numbers
  use blha_config
  use blha_interface
```

⟨*Standard module head*⟩

⟨*BLHA driver: public*⟩

```
contains
```

⟨*BLHA driver: procedures*⟩

```
end module blha_driver
```

21.1.1 Configuration

Parameters to enumerate the different options in the order.

⟨*BLHA config: parameters*⟩≡
```
    integer, public, parameter :: &
        BLHA_MEST_SUM=1, BLHA_MEST_AVG=2, BLHA_MEST_OTHER=3
    integer, public, parameter :: &
        BLHA_CT_QCD=1, BLHA_CT_EW=2, BLHA_CT_QED=3, BLHA_CT_OTHER=4
    integer, public, parameter :: &
        BLHA_IRREG_CDR=1, BLHA_IRREG_DRED=2, BLHA_IRREG_THV=3, &
        BLHA_IRREG_MREG=4, BLHA_IRREG_OTHER=5
    integer, public, parameter :: &
        BLHA_SUBMODE_NONE = 1, BLHA_SUBMODE_OTHER = 2
    integer, public, parameter :: &
        BLHA_MPS_ONSHELL=1, BLHA_MPS_OTHER=2
    integer, public, parameter :: &
        BLHA_MODE_GOSAM=1, BLHA_MODE_FEYNARTS = 2, BLHA_MODE_GENERIC=3
    integer, public, parameter :: &
        BLHA_OM_NONE=1, BLHA_OM_NOCPL=2, BLHA_OM_OTHER=3
```

This type encapsulates a BLHA request.

⟨*BLHA config: public*⟩≡
```
    public :: blha_configuration_t
    public :: blha_cfg_process_node_t
```

⟨*BLHA config: types*⟩≡
```
    type :: blha_cfg_process_node_t
        integer, dimension(:), allocatable :: pdg_in, pdg_out
```

```
        integer, dimension(:), allocatable :: fingerprint
        integer :: nsub
        integer, dimension(:), allocatable :: ids
        type(blha_cfg_process_node_t), pointer :: next => null ()
     end type blha_cfg_process_node_t

     type :: blha_configuration_t
        type(string_t) :: name
        type(model_t), pointer :: model
        type(string_t) :: md5
        logical :: dirty = .true.
        integer :: n_proc = 0
        integer :: mode = BLHA_MODE_GENERIC
        type(blha_cfg_process_node_t), pointer :: processes => null ()
        integer, dimension(2) :: matrix_element_square_type = BLHA_MEST_SUM
        type(string_t), dimension (2) :: matrix_element_square_type_other
        integer :: correction_type = BLHA_CT_QCD
        type(string_t) :: correction_type_other
        integer :: irreg = BLHA_IRREG_THV
        type(string_t) :: irreg_other
        integer :: massive_particle_scheme = BLHA_MPS_ONSHELL
        type(string_t) :: massive_particle_scheme_other
        integer :: subtraction_mode = BLHA_SUBMODE_NONE
        type(string_t) :: subtraction_mode_other
        type(string_t) :: model_file
        logical :: subdivide_subprocesses = .false.
        integer :: alphas_power = -1, alpha_power = -1
        integer :: operation_mode = BLHA_OM_NONE
        type(string_t) :: operation_mode_other
     end type blha_configuration_t
```

Initializer.

⟨*BLHA config: public*⟩+≡

```
     public :: blha_configuration_init
```

⟨*BLHA config: procedures*⟩≡

```
     subroutine blha_configuration_init (cfg, name, model, mode)
       type(blha_configuration_t), intent(out) :: cfg
       type(string_t), intent(in) :: name
       type(model_t), target, intent(in) :: model
       integer, intent(in), optional :: mode
       cfg%name = name
       cfg%model => model
       if (present (mode)) cfg%mode = mode
     end subroutine blha_configuration_init
```

Finalizer.

⟨*BLHA config: public*⟩+≡

```
     public :: blha_configuration_final
```

⟨*BLHA config: procedures*⟩+≡

```
     subroutine blha_configuration_final (cfg)
       type(blha_configuration_t), intent(inout) :: cfg
       type(blha_cfg_process_node_t), pointer :: cur, next
```

```
      cur => cfg%processes
      do while (associated (cur))
         next => cur%next
         deallocate (cur)
         nullify (cur)
         cur => next
      end do
   end subroutine blha_configuration_final
```

Merge sort a process list w.r.t. to the process fingerprints. This is necessary for canonicalizing the process list prior to calculating the MD5 sum.

⟨*BLHA config: procedures*⟩+≡

```
   subroutine sort_processes (list, n)
     type(blha_cfg_process_node_t), pointer :: list
     integer, intent(in), optional :: n
     type :: pnode
        type(blha_cfg_process_node_t), pointer :: p
     end type pnode
     type(pnode), dimension(:), allocatable :: array
     integer :: count, i, s, i1, i2, i3
     type(blha_cfg_process_node_t), pointer :: node
     if (present (n)) then
        count = n
     else
        node => list
        count = 0
        do while (associated (node))
           node => node%next
           count = count + 1
        end do
     end if
     ! Store list nodes into an array
     if (count == 0) return
     allocate (array(count))
     i = 1
     node => list
     do i = 1, count
        array(i)%p => node
        node => node%next
     end do
     s = 1
     ! Merge sort the array
     do while (s < count)
        i = 0
        i1 = 1
        i2 = s
        do while (i2 < count)
           i3 = min (s*(i+2), count)
           array(i1:i3) = merge (array(i1:i2), array(i2+1:i3))
           i = i + 2
           i1 = s*i+1
           i2 = s*(i+1)
        end do
```

```
        s = s * 2
     end do
     ! Relink according to their new order
     list => array(1)%p
     nullify (array(count)%p%next)
     node => list
     do i = 2, count
        node%next => array(i)%p
        node => node%next
     end do

contains

   ! .le. comparision
   function lt (n1, n2) result (predicate)
      type(blha_cfg_process_node_t), intent(in) :: n1, n2
      logical :: predicate
      integer :: i
      predicate = .true.
      do i = 1, size (n1%fingerprint)
         if (n1%fingerprint(i) < n2%fingerprint(i)) return
         if (n1%fingerprint(i) > n2%fingerprint(i)) then
            predicate = .false.
            return
         end if
      end do
   end function lt

   ! Sorting core --- merge two sorted chunks
   function merge (l1, l2) result (lo)
      type(pnode), dimension(:), intent(in) :: l1, l2
      type(pnode), dimension(size (l1) + size (l2)) :: lo
      integer :: i, i1, i2
      i1 = 1
      i2 = 1
      do i = 1, size (lo)
         if (i1 > size (l1)) then
            lo(i)%p => l2(i2)%p
            i2 = i2 + 1
         elseif (i2 > size (l2)) then
            lo(i)%p => l1(i1)%p
            i1 = i1 + 1
         elseif (lt (l1(i1)%p, l2(i2)%p)) then
            lo(i)%p => l1(i1)%p
            i1 = i1 + 1
         else
            lo(i)%p => l2(i2)%p
            i2 = i2 + 1
         end if
      end do
   end function merge

end subroutine sort_processes
```

Append a process. This expands the flavor sum, sorts it and then eliminates any duplicates.

⟨*BLHA config: public*⟩+≡
```
public :: blha_configuration_append_process
```

⟨*BLHA config: procedures*⟩+≡
```
subroutine blha_configuration_append_process (cfg, pdg_in, pdg_out, nsub, ids)
  type(blha_configuration_t), intent(inout) :: cfg
  type(pdg_array_t), dimension(:), intent(in) :: pdg_in, pdg_out
  integer, optional, intent(in) :: nsub
  integer, optional, dimension(:), intent(in) :: ids
  type(blha_cfg_process_node_t), pointer :: root, node, tmp
  ! Multiindex for counting through the PDG numbers
  integer, dimension(size (pdg_in)) :: i_in
  integer, dimension(size (pdg_out)) :: i_out
  ! Handle the list of lists
  type :: ilist
     integer, dimension(:), allocatable :: i
  end type ilist
  type(ilist), dimension(size (pdg_in)) :: ilist_i
  type(ilist), dimension(size (pdg_out)) :: ilist_o
  integer :: i, j, nproc
  logical :: inc
  ! Extract PDGs into integer lists
  do i = 1, size (pdg_in)
     ilist_i(i)%i = pdg_in(i)
  end do
  do i = 1, size (pdg_out)
     ilist_o(i)%i = pdg_out(i)
  end do
  i_in = 1
  i_out = 1
  allocate (root)
  node => root
  ! Perform the expansion
  nproc = 0
  EXPAND: do
     ! Transfer the PDG selection...
     allocate (node%pdg_in(size (pdg_in)))
     allocate (node%pdg_out(size (pdg_out)))
     allocate (node%fingerprint (size (pdg_in) + size (pdg_out)))
     if (present (nsub)) node%nsub = nsub
     if (present (ids)) then
        allocate (node%ids(size (ids)))
        node%ids = ids
     end if
     forall (j=1:size(ilist_i)) &
        node%pdg_in(j) = ilist_i(j)%i(i_in(j))
     forall (j=1:size(ilist_o)) &
        node%pdg_out(j) = ilist_o(j)%i(i_out(j))
     node%fingerprint = [ node%pdg_in, sort (node%pdg_out) ]
     nproc = nproc + 1
     inc = .false.
     ! ... and increment the multiindex
```

```
      do j = 1, size (i_out)
         if (i_out(j) < size (ilist_o(j)%i)) then
            i_out(j) = i_out(j) + 1
            inc = .true.
            exit
         else
            i_out(j) = 1
         end if
      end do
      if (.not. inc) then
         do j = 1, size (i_in)
            if (i_in(j) < size (ilist_i(j)%i)) then
               i_in(j) = i_in(j) + 1
               inc = .true.
               exit
            else
               i_in(j) = 1
            end if
         end do
      end if
      if (.not. inc) exit EXPAND
      allocate (node%next)
      node => node%next
   end do EXPAND
   ! Do the sorting
   call sort_processes (root, nproc)
   ! Kill duplicates
   node => root
   do while (associated (node))
      if (.not. associated (node%next)) exit
      if (all (node%fingerprint == node%next%fingerprint)) then
         tmp => node%next%next
         deallocate (node%next)
         node%next => tmp
         nproc = nproc - 1
      else
         node => node%next
      end if
   end do
   ! Append the remaining list
   if (associated (cfg%processes)) then
      node => cfg%processes
      do while (associated (node%next))
         node => node%next
      end do
      node%next => root
   else
      cfg%processes => root
   end if
   cfg%n_proc = cfg%n_proc + nproc
   cfg%dirty = .true.

 end subroutine blha_configuration_append_process
```

Change parameter(s).

⟨*BLHA config: public*⟩+≡
```
    public :: blha_configuration_set
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine blha_configuration_set ( cfg, &
         matrix_element_square_type_hel, matrix_element_square_type_hel_other, &
         matrix_element_square_type_col, matrix_element_square_type_col_other, &
         correction_type, correction_type_other, &
         irreg, irreg_other, &
         massive_particle_scheme, massive_particle_scheme_other, &
         subtraction_mode, subtraction_mode_other, &
         model_file, subdivide_subprocesses, alphas_power, alpha_power, &
         operation_mode, operation_mode_other)
      type(blha_configuration_t), intent(inout) :: cfg
      integer, optional, intent(in) :: matrix_element_square_type_hel
      type(string_t), optional, intent(in) :: matrix_element_square_type_hel_other
      integer, optional, intent(in) :: matrix_element_square_type_col
      type(string_t), optional, intent(in) :: matrix_element_square_type_col_other
      integer, optional, intent(in) :: correction_type
      type(string_t), optional, intent(in) :: correction_type_other
      integer, optional, intent(in) :: irreg
      type(string_t), optional, intent(in) :: irreg_other
      integer, optional, intent(in) :: massive_particle_scheme
      type(string_t), optional, intent(in) :: massive_particle_scheme_other
      integer, optional, intent(in) :: subtraction_mode
      type(string_t), optional, intent(in) :: subtraction_mode_other
      type(string_t), optional, intent(in) :: model_file
      logical, optional, intent(in) :: subdivide_subprocesses
      integer, intent(in), optional :: alphas_power, alpha_power
      integer, intent(in), optional :: operation_mode
      type(string_t), intent(in), optional :: operation_mode_other
      if (present (matrix_element_square_type_hel)) &
         cfg%matrix_element_square_type(1) = matrix_element_square_type_hel
      if (present (matrix_element_square_type_hel_other)) &
         cfg%matrix_element_square_type_other(1) = matrix_element_square_type_hel_other
      if (present (matrix_element_square_type_col)) &
         cfg%matrix_element_square_type(2) = matrix_element_square_type_col
      if (present (matrix_element_square_type_col_other)) &
         cfg%matrix_element_square_type_other(2) = matrix_element_square_type_col_other
      if (present (correction_type)) &
         cfg%correction_type = correction_type
      if (present (correction_type_other)) &
         cfg%correction_type_other = correction_type_other
      if (present (irreg)) &
         cfg%irreg = irreg
      if (present (irreg_other)) &
         cfg%irreg_other = irreg_other
      if (present (massive_particle_scheme)) &
         cfg%massive_particle_scheme = massive_particle_scheme
      if (present (massive_particle_scheme_other)) &
         cfg%massive_particle_scheme_other = massive_particle_scheme_other
      if (present (subtraction_mode)) &
         cfg%subtraction_mode = subtraction_mode
      if (present (subtraction_mode_other)) &
```

```
       cfg%subtraction_mode_other = subtraction_mode_other
    if (present (model_file)) &
       cfg%model_file = model_file
    if (present (subdivide_subprocesses)) &
       cfg%subdivide_subprocesses = subdivide_subprocesses
    if (present (alphas_power)) &
       cfg%alphas_power = alphas_power
    if (present (alpha_power)) &
       cfg%alpha_power = alpha_power
    if (present (operation_mode)) &
       cfg%operation_mode = operation_mode
    if (present (operation_mode_other)) &
       cfg%operation_mode_other = operation_mode_other
    cfg%dirty = .true.
  end subroutine blha_configuration_set
```

Write the BLHA file. Internal mode is intented for md5summing only.

⟨*BLHA config: public*⟩+≡
```
    public :: blha_configuration_write
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine blha_configuration_write (cfg, unit, internal)
      type(blha_configuration_t), intent(in) :: cfg
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: internal
      integer :: u
      logical :: full
      type(string_t) :: buf
      type(blha_cfg_process_node_t), pointer :: node
      integer :: i
      character(3) :: pdg_char
      character(len=25), parameter :: pad = ""
      u = output_unit (unit)
      full = .true.; if (present (internal)) full = .not. internal
      if (full .and. cfg%dirty) call msg_bug ( &
         "BUG: attempted to write out a dirty BLHA configuration")
      if (full) then
         write (u,'(A)') "# BLHA order written by WHIZARD ⟨Version⟩"
         write (u,'(A)')
      end if
      select case (cfg%mode)
         case (BLHA_MODE_GOSAM); buf = "GoSam"
         case default; buf = "vanilla"
      end select
      write (u,'(A)') "# BLHA interface mode: " // char (buf)
      write (u,'(A)') "# process: " // char (cfg%name)
      write (u,'(A)') "# model: " // char (cfg%model%get_name ())
      if (full) then
         write (u,'(A)')
         write (u,'(A)') '#@WO MD5 "' // char (cfg%md5) // '"'
         write (u,'(A)')
      end if
      if (all (cfg%matrix_element_square_type == BLHA_MEST_SUM)) then
         buf = "CHsummed"
```

```
elseif (all (cfg%matrix_element_square_type == BLHA_MEST_AVG)) then
   buf = "CHaveraged"
else
   buf = (render_mest ("H", cfg%matrix_element_square_type(1), &
         cfg%matrix_element_square_type_other(1)) // " ") // &
      render_mest ("C", cfg%matrix_element_square_type(2), &
         cfg%matrix_element_square_type_other(2))
end if
write (u,'(A25,A)') "MatrixElementSquareType" // pad, char (buf)
select case (cfg%correction_type)
   case (BLHA_CT_QCD); buf = "QCD"
   case (BLHA_CT_EW); buf = "EW"
   case (BLHA_CT_QED); buf = "QED"
   case default; buf = cfg%correction_type_other
end select
write (u,'(A25,A)') "CorrectionType" // pad, char (buf)
select case (cfg%irreg)
   case (BLHA_IRREG_CDR); buf = "CDR"
   case (BLHA_IRREG_DRED); buf = "DRED"
   case (BLHA_IRREG_THV); buf = "tHV"
   case (BLHA_IRREG_MREG); buf = "MassReg"
   case default; buf = cfg%irreg_other
end select
write (u,'(A25,A)') "IRregularisation" // pad, char (buf)
select case (cfg%massive_particle_scheme)
   case (BLHA_MPS_ONSHELL); buf = "OnShell"
   case default; buf = cfg%massive_particle_scheme_other
end select
write (u,'(A25,A)') "MassiveParticleScheme" // pad, char (buf)
select case (cfg%subtraction_mode)
   case (BLHA_SUBMODE_NONE); buf = "None"
   case default; buf = cfg%subtraction_mode_other
end select
write (u,'(A25,A)') "IRsubtractionMethod" // pad, char (buf)
write (u,'(A25,A)') "ModelFile" // pad, char (cfg%model_file)
if (cfg%subdivide_subprocesses) then
   write (u,'(A25,A)') "SubdivideSubprocesses" // pad, "yes"
else
   write (u,'(A25,A)') "SubdivideSubprocess" // pad, "no"
end if
if (cfg%alphas_power >= 0) write (u,'(A25,A)') &
   "AlphasPower" // pad, int2char (cfg%alphas_power)
if (cfg%alpha_power >= 0) write (u,'(A25,A)') &
   "AlphaPower " // pad, int2char (cfg%alpha_power)
if (full) then
   write (u,'(A)')
   write (u,'(A)') "# Process definitions"
   write (u,'(A)')
end if
node => cfg%processes
do while (associated (node))
   buf = ""
   do i = 1, size (node%pdg_in)
      write (pdg_char,'(I3)') node%pdg_in(i)
```

```
              buf = (buf // pdg_char) // " "
           end do
           buf = buf // "-> "
           do i = 1, size (node%pdg_out)
              write (pdg_char,'(I3)') node%pdg_out(i)
              buf = (buf // pdg_char) // " "
           end do
           write (u,'(A)') char (trim (buf))
           node => node%next
        end do

    contains

      function render_mest (prefix, mest, other) result (tag)
        character, intent(in) :: prefix
        integer, intent(in) :: mest
        type(string_t), intent(in) :: other
        type(string_t) :: tag
        select case (mest)
        case (BLHA_MEST_AVG); tag = prefix // "averaged"
        case (BLHA_MEST_SUM); tag = prefix // "summed"
        case default; tag = other
        end select
      end function render_mest

    end subroutine blha_configuration_write
```

"Freeze" the configuration by calculating the MD5 sum.

⟨*BLHA config: public*⟩+≡
```
    public :: blha_configuration_freeze
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine blha_configuration_freeze (cfg)
      type(blha_configuration_t), intent(inout) :: cfg
      integer :: u
      if (.not. cfg%dirty) return
      call sort_processes (cfg%processes)
      u = free_unit ()
      open (unit=u, status="scratch", action="readwrite")
      call blha_configuration_write (cfg, u, internal=.true.)
      rewind (u)
      cfg%md5 = md5sum (u)
      cfg%dirty = .false.
      close (u)
    end subroutine blha_configuration_freeze
```

Read a contract file, again creating a `blha_configuration_t` object.

⟨*BLHA config: public*⟩+≡
```
    public :: blha_read_contract
```

⟨*BLHA config: interfaces*⟩≡
```
    interface blha_read_contract
       module procedure blha_read_contract_unit, &
            blha_read_contract_file
```

```
        end interface blha_read_contract
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine blha_read_contract_file (cfg, ok, fname, success)
      type(blha_configuration_t), intent(inout) :: cfg
      logical, intent(out) :: ok
      type(string_t), intent(in) :: fname
      logical, intent(out), optional :: success
      integer :: u, stat
      u = free_unit ()
      open (u, file=char (fname), status="old", action="read", iostat=stat)
      if (stat /= 0) then
         if (present (success)) then
            success = .false.
            return
         else
            call msg_bug ('Unable to open contract file "' // char (fname) // '"')
         end if
      end if
      call blha_read_contract_unit (cfg, ok, u, success)
      close (u)
    end subroutine blha_read_contract_file

    subroutine blha_read_contract_unit (cfg, ok, u, success)
      type(blha_configuration_t), intent(inout) :: cfg
      logical, intent(out) :: ok
      integer, intent(in) :: u
      logical, intent(out), optional :: success
      type(stream_t) :: stream
      type(ifile_t) :: preprocessed
      type(lexer_t) :: lexer
      type(parse_tree_t) :: parse_tree
      type(string_t) :: md5
      call stream_init (stream, u)
      call contract_preprocess (stream, preprocessed)
      call stream_final (stream)
      call stream_init (stream, preprocessed)
      call blha_init_lexer (lexer)
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (parse_tree, syntax_blha_contract, lexer)
      call blha_transfer_contract (cfg, ok, parse_tree, success)
      call blha_configuration_write (cfg, internal=.true.)
      call lexer_final (lexer)
      call stream_final (stream)
      call ifile_final (preprocessed)
      if (ok) then
         md5 = cfg%md5
         call blha_configuration_freeze (cfg)
         if (char (trim (md5 )) /= "") then
            if (md5 /= cfg%md5) then
               call msg_warning ("BLHA contract does not match the recorded " &
                  // "checksum --- this counts as an error!")
               ok = .false.
            end if
```

```
            else
                call msg_warning ("It seems the OLP scrubbed our checksum, unable " &
                    // "to check contract consistency.")
            end if
        end if
    end subroutine blha_read_contract_unit
```

Walk the parse tree and transfer the results to the `blha_configuration` object. The `goto` is a poor man's replacement for exceptions which would be an appropiate error handling mechanism here.

⟨*BLHA config: procedures*⟩+≡

```
    subroutine blha_transfer_contract (cfg, ok, parse_tree, success)
        type(blha_configuration_t), intent(inout) :: cfg
        logical, intent(out) :: ok
        type(parse_tree_t), intent(in), target :: parse_tree
        logical, intent(out), optional :: success
        type(parse_node_t), pointer :: pn_root, pn_line, pn_request, &
            pn_result, pn_key, pn_opt, pn_state_in, pn_state_out, pn_pdg
        type(string_t) :: emsg
        integer :: nopt, i, nsub
        integer, dimension(:), allocatable :: ids
        logical, dimension(2) :: flags
        type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out
        ok = .true.
        pn_root => parse_tree_get_root_ptr (parse_tree)
        pn_line => parse_node_get_sub_ptr (pn_root)
        do while (associated (pn_line))
            pn_request => parse_node_get_sub_ptr (pn_line)
            if (.not. associated (pn_request)) cycle
            if (char (parse_node_get_rule_key (pn_request)) == "process") then
                pn_result => parse_node_get_sub_ptr (pn_line, 2)
                pn_state_in => parse_node_get_sub_ptr (pn_request, 1)
                pn_state_out => parse_node_get_sub_ptr (pn_request, 3)
                allocate (pdg_in (parse_node_get_n_sub (pn_state_in)))
                allocate (pdg_out (parse_node_get_n_sub (pn_state_out)))
                i = 1
                pn_pdg => parse_node_get_sub_ptr (pn_state_in)
                do while (associated (pn_pdg))
                    pdg_in(i) = [get_int (pn_pdg)]
                    pn_pdg => parse_node_get_next_ptr (pn_pdg)
                    i = i + 1
                end do
                i = 1
                pn_pdg => parse_node_get_sub_ptr (pn_state_out)
                do while (associated (pn_pdg))
                    pdg_out(i) = [get_int (pn_pdg)]
                    pn_pdg => parse_node_get_next_ptr (pn_pdg)
                    i = i + 1
                end do
                i = parse_node_get_n_sub (pn_result)
                emsg = "broken process line"
                if (i < 2) goto 10
                pn_opt => parse_node_get_sub_ptr (pn_result, 2)
```

```
                 do while (associated (pn_opt))
                    if (char (parse_node_get_rule_key (pn_opt)) == "string") then
                       call msg_warning ("While reading the BLHA contract: " // &
                          'the OLP returned an error for a process: "' // &
                          char (parse_node_get_string (pn_opt)) // '"')
                       ok = .false.
                       return
                    end if
                    pn_opt => parse_node_get_next_ptr (pn_opt)
                 end do
                 pn_opt => parse_node_get_sub_ptr (pn_result, 2)
                 nsub = get_int (pn_opt)
                 if (nsub /= i - 2) goto 10
                 allocate (ids(nsub))
                 i = 1
                 pn_opt => parse_node_get_next_ptr (pn_opt)
                 do while (associated (pn_opt))
                    ids(i) = get_int (pn_opt)
                    pn_opt => parse_node_get_next_ptr (pn_opt)
                 end do
                 call blha_configuration_append_process (cfg, pdg_in, pdg_out, &
                    nsub=nsub, ids=ids)
                 deallocate (pdg_in, pdg_out, ids)
              else
                 pn_result => parse_node_get_sub_ptr (parse_node_get_next_ptr (pn_request), 2)
                 pn_key => parse_node_get_sub_ptr (pn_request)
                 pn_opt => parse_node_get_next_ptr (pn_key)
                 nopt = parse_node_get_n_sub (pn_request) - 1
                 select case (char (parse_node_get_rule_key (pn_key)))
                 case ("md5")
                    cfg%md5 = parse_node_get_string (pn_opt)
                 case ("modelfile")
                    cfg%model_file = get_fname (pn_opt)
                    call check_result (pn_result, "ModelFile")
                 case ("irregularisation")
                    select case (lower_case (char (parse_node_get_string (pn_opt))))
                       case ("cdr"); cfg%irreg = BLHA_IRREG_CDR
                       case ("dred"); cfg%irreg = BLHA_IRREG_DRED
                       case ("thv"); cfg%irreg = BLHA_IRREG_THV
                       case ("mreg"); cfg%irreg = BLHA_IRREG_MREG
                       case default
                          cfg%irreg = BLHA_IRREG_OTHER
                          cfg%irreg_other = parse_node_get_string (pn_opt)
                    end select
                    call check_result (pn_result, "IRRegularisation")
                 case ("irsubtractionmethod")
                    select case (lower_case (char (parse_node_get_string (pn_opt))))
                       case ("none"); cfg%subtraction_mode = BLHA_SUBMODE_NONE
                       case default
                          cfg%subtraction_mode = BLHA_SUBMODE_OTHER
                          cfg%subtraction_mode_other = parse_node_get_string(pn_opt)
                    end select
                    call check_result (pn_result, "IRSubtractionMethod")
                 case ("massiveparticlescheme")
```

```fortran
         select case (lower_case (char (parse_node_get_string (pn_opt))))
            case ("onshell")
               cfg%massive_particle_scheme = BLHA_MPS_ONSHELL
            case default
               cfg%massive_particle_scheme = BLHA_MPS_OTHER
               cfg%massive_particle_scheme_other = &
                  parse_node_get_string (pn_opt)
         end select
         call check_result (pn_result, "MassiveParticleScheme")
      case ("matrixelementsquaretype")
         select case (nopt)
            case (1)
               select case (lower_case (char (parse_node_get_string (pn_opt))))
                  case ("chsummed")
                     cfg%matrix_element_square_type = BLHA_MEST_SUM
                  case ("chaveraged")
                     cfg%matrix_element_square_type = BLHA_MEST_AVG
                  case default
                     emsg = "invalid MatrixElementSquareType: " // &
                        parse_node_get_string (pn_opt)
                     goto 10
               end select
            case (2)
               do i = 1, 2
                  pn_opt => parse_node_get_next_ptr (pn_key, i)
                  select case (lower_case (char (parse_node_get_string ( &
                     pn_opt))))
                     case ("csummed")
                        cfg%matrix_element_square_type(2) = BLHA_MEST_SUM
                        flags(2) = .true.
                     case ("caveraged")
                        cfg%matrix_element_square_type(2) = BLHA_MEST_AVG
                        flags(2) = .true.
                     case ("hsummed")
                        cfg%matrix_element_square_type(1) = BLHA_MEST_SUM
                        flags(1) = .true.
                     case ("haveraged")
                        cfg%matrix_element_square_type(1) = BLHA_MEST_AVG
                        flags(1) = .true.
                     case default
                        emsg = "invalid MatrixElementSquareType: " // &
                           parse_node_get_string (pn_opt)
                        goto 10
                  end select
               end do
               if (.not. all (flags)) then
                  emsg = "MatrixElementSquareType: setup not exhaustive"
                  goto 10
               end if
            case default
               emsg = "MatrixElementSquareType: too many options"
               goto 10
         end select
         call check_result (pn_result, "MatrixElementSquareType")
```

2757

```
           case ("correctiontype")
              select case (lower_case (char (parse_node_get_string (pn_opt))))
                 case ("qcd"); cfg%correction_type = BLHA_CT_QCD
                 case ("qed"); cfg%correction_type = BLHA_CT_QED
                 case ("ew"); cfg%correction_type = BLHA_CT_EW
                 case default
                    cfg%correction_type = BLHA_CT_OTHER
                    cfg%correction_type_other = parse_node_get_string (pn_opt)
              end select
              call check_result (pn_result, "CorrectionType")
           case ("alphaspower")
              cfg%alphas_power = get_int (pn_opt)
              call check_result (pn_result, "AlphasPower")
           case ("alphapower")
              cfg%alpha_power = get_int (pn_opt)
              call check_result (pn_result, "AlphaPower")
           case ("subdividesubprocess")
              select case (lower_case (char (parse_node_get_string (pn_opt))))
                 case ("yes"); cfg%subdivide_subprocesses = .true.
                 case ("no"); cfg%subdivide_subprocesses = .false.
                 case default
                    emsg = 'SubdivideSubprocess: invalid argument "' // &
                       parse_node_get_string (pn_opt) // '"'
                    goto 10
              end select
              call check_result (pn_result, "SubdivideSubprocess")
           case default
              emsg = "unknown statement: " // parse_node_get_rule_key (pn_key)
              goto 10
        end select
     end if
     pn_line => parse_node_get_next_ptr (pn_line)
  end do
  if (present (success)) success = .true.
  return
10  continue
  if (present (success)) then
     call msg_error ("Error reading BLHA contract: " // char (emsg))
     success = .false.
     return
  else
     call msg_fatal ("Error reading BLHA contract: " // char (emsg))
  end if

contains

  function get_int (pn) result (i)
    type(parse_node_t), pointer :: pn
    integer :: i
    if (char (parse_node_get_rule_key (pn)) == "integer") then
       i = parse_node_get_integer (pn)
    else
       i = parse_node_get_integer (parse_node_get_sub_ptr (pn, 2))
       if (char (parse_node_get_rule_key (parse_node_get_sub_ptr (pn))) &
```

```
                    == "-") i = -i
         end if
      end function get_int

      subroutine check_result (pn, step)
        type(parse_node_t), pointer :: pn
        character(*), intent(in) :: step
        type(string_t) :: res
        res = parse_node_get_string (pn)
        if (char (trim (res)) == "") then
           call msg_warning ("BLHA contract file: " // step // &
                ": OLP didn't return a status --- assuming an error")
           ok = .false.
        elseif (char (upper_case (res)) /= "OK") then
           call msg_warning ("BLHA contract file: " // step // &
                ': OLP error "' // char (res) // '"')
           ok = .false.
        end if
      end subroutine check_result

      function get_fname (pn) result (fname)
        type(parse_node_t), pointer :: pn
        type(string_t) :: fname
        type(parse_node_t), pointer :: pn_component
        if (char (parse_node_get_rule_key (pn)) == "string") then
           fname = parse_node_get_string (pn)
        else
           fname = ""
           pn_component => parse_node_get_sub_ptr (pn)
           do while (associated (pn_component))
              if (char (parse_node_get_rule_key (pn_component)) == "id") then
                 fname = fname // parse_node_get_string (pn_component)
              else
                 fname = fname // parse_node_get_key (pn_component)
              end if
              pn_component => parse_node_get_next_ptr (pn_component)
           end do
        end if
      end function get_fname

  end subroutine blha_transfer_contract
```

Initialize the lexer.
⟨*BLHA config: procedures*⟩+≡
```
  subroutine blha_init_lexer (lexer)
    type(lexer_t), intent(inout) :: lexer
    call lexer_init (lexer, &
       comment_chars = "#", &
       quote_chars = '"', &
       quote_match = '"', &
       single_chars = '{}|./\:', &
       special_class = ["->"], &
       keyword_list = syntax_get_keyword_list_ptr (syntax_blha_contract), &
       upper_case_keywords = .false. &
```

```
      )
    end subroutine blha_init_lexer
```

Define the parser syntax table.

⟨*BLHA config: variables*⟩≡
```
    type(syntax_t), target, save :: syntax_blha_contract
```

⟨*BLHA config: public*⟩+≡
```
    public :: syntax_blha_contract_init
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine syntax_blha_contract_init ()
      type(ifile_t) :: ifile
      call ifile_append (ifile, "SEQ contract = line*")
      call ifile_append (ifile, "KEY '->'")
      call ifile_append (ifile, "KEY '.'")
      call ifile_append (ifile, "KEY '/'")
      call ifile_append (ifile, "KEY '\'")
      call ifile_append (ifile, "KEY '+'")
      call ifile_append (ifile, "KEY '-'")
      call ifile_append (ifile, "KEY '|'")
      call ifile_append (ifile, "KEY ':'")
      call ifile_append (ifile, "IDE id")
      call ifile_append (ifile, "INT integer")
      call ifile_append (ifile, "ALT sign = '+' | '-'")
      call ifile_append (ifile, "SEQ signed_integer = sign integer")
      call ifile_append (ifile, "QUO string = '"'...'"'")
      call ifile_append (ifile, "GRO line = '{' line_contents '}'")
      call ifile_append (ifile, "SEQ line_contents = request result?")
      call ifile_append (ifile, "ALT request = definition | process")
      call ifile_append (ifile, "ALT definition = option_unary | option_nary | " &
         // "option_path | option_numeric")
      call ifile_append (ifile, "KEY matrixelementsquaretype")
      call ifile_append (ifile, "KEY irregularisation")
      call ifile_append (ifile, "KEY massiveparticlescheme")
      call ifile_append (ifile, "KEY irsubtractionmethod")
      call ifile_append (ifile, "KEY modelfile")
      call ifile_append (ifile, "KEY operationmode")
      call ifile_append (ifile, "KEY subdividesubprocess")
      call ifile_append (ifile, "KEY alphaspower")
      call ifile_append (ifile, "KEY alphapower")
      call ifile_append (ifile, "KEY correctiontype")
      call ifile_append (ifile, "KEY md5")
      call ifile_append (ifile, "SEQ option_unary = key_unary arg")
      call ifile_append (ifile, "SEQ option_nary = key_nary arg+")
      call ifile_append (ifile, "SEQ option_path = key_path arg_path")
      call ifile_append (ifile, "SEQ option_numeric = key_numeric arg_numeric")
      call ifile_append (ifile, "ALT key_unary = irregularisation | " &
         // "massiveparticlescheme | irsubtractionmethod | subdividesubprocess | " &
         // "correctiontype | md5")
      call ifile_append (ifile, "ALT key_nary = matrixelementsquaretype | " &
         // "operationmode")
      call ifile_append (ifile, "ALT key_numeric = alphaspower | alphapower")
      call ifile_append (ifile, "ALT key_path = modelfile")
      call ifile_append (ifile, "ALT arg = id | string")
```

```
      call ifile_append (ifile, "ALT arg_numeric = integer | signed_integer")
      call ifile_append (ifile, "ALT arg_path = filename | string")
      call ifile_append (ifile, "SEQ filename = filename_atom+")
      call ifile_append (ifile, "ALT filename_atom = id | '.' | '/' | '\' | ':'")
      call ifile_append (ifile, "SEQ process = state '->' state")
      call ifile_append (ifile, "SEQ state = pdg+")
      call ifile_append (ifile, "ALT pdg = integer | signed_integer")
      call ifile_append (ifile, "SEQ result = '|' result_atom+")
      call ifile_append (ifile, "ALT result_atom = integer | string")
      call syntax_init (syntax_blha_contract, ifile)
      call ifile_final (ifile)
    end subroutine syntax_blha_contract_init
```

⟨*BLHA config: public*⟩+≡
```
    public :: syntax_blha_contract_final
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine syntax_blha_contract_final
      call syntax_final (syntax_blha_contract)
    end subroutine syntax_blha_contract_final
```

As the contract file is line-oriented, we apply a preprocessing step which reformats the file in a way suitable for our free-form parser.

⟨*BLHA config: procedures*⟩+≡
```
    subroutine contract_preprocess (stream, ifile)
      type(stream_t), intent(inout) :: stream
      type(ifile_t), intent(out) :: ifile
      type(string_t) :: buf, reg, transformed
      integer :: stat, n
      buf = ""
      LINES: do
        call stream_get_record (stream, reg, stat)
        select case (stat)
          case (0)
          case (EOF); exit LINES
          case default
            call msg_bug ("I/O error while reading BLHA contract file")
        end select
        buf = buf // trim (reg)
        ! Take care of continuation lines
        if (char (extract (buf, len (buf), len(buf))) == '&') then
          buf = extract (buf, 1, len (buf) - 1) // " "
          cycle LINES
        end if
        buf = adjustl (buf)
        ! Transform #@WO comments into ordinary statements
        if (char (extract (buf, 1, 4)) == "#@WO") &
          buf = extract (buf, 5)
        ! Kill comments and blank lines
        if ((char (trim (buf)) == "") .or. &
          (char (extract (buf, 1, 1)) == "#")) then
            buf = ""
            cycle LINES
```

2761

```
         end if
         ! Chop off any end-of-line comments
         call split (buf, reg, "#")
         ! Split line into order and result
         call split (reg, buf, "|")
         reg = trim (adjustl (reg))
         buf = trim (adjustl (buf))
         ! Check whether the order is a process definition
         n = scan (buf, ">")
         if (n == 0) then
            ! No -> quote result
            reg = ('"' // reg) // '"'
         else
            ! Yes -> leave any numbers as they are, quote any leftovers
            n = scan (reg, "0123456789", back=.true.)
            if (n < len (reg)) &
               reg = char (extract (reg, 1, n)) // ' "' // &
                  char (trim (adjustl (extract (reg, n+1)))) // '"'
         end if
         ! Enclose the line into curly brackets
         transformed = "{" // char (buf) // " | " // char (reg) // "}"
         call ifile_append (ifile, transformed)
         buf = ""
      end do LINES
    end subroutine contract_preprocess
```

Test.

⟨*BLHA config: public*⟩+≡
```
    public :: blha_config_test
```

⟨*BLHA config: procedures*⟩+≡
```
    subroutine blha_config_test (model, cfg, ok)
      type(pdg_array_t), dimension(2) :: pdg_in
      type(pdg_array_t), dimension(4) :: pdg_out
      type(model_t), pointer :: model
      type(blha_configuration_t), intent(out) :: cfg
      logical, intent(out) :: ok
      integer :: u
      logical :: flag
      ok = .false.
      pdg_in(1) = [1, 2, -1, -2]
      pdg_in(2) = pdg_in(1)
      pdg_out(1) = pdg_in(1)
      pdg_out(2) = [11]
      pdg_out(3) = [-11]
      pdg_out(4) = pdg_out(1)
      call blha_configuration_init (cfg, var_str ("test"), model)
      call blha_configuration_set (cfg, alphas_power = 2, alpha_power = 3)
      call blha_configuration_append_process (cfg, pdg_in, pdg_out)
      call blha_configuration_freeze (cfg)
      print *
      call blha_configuration_write (cfg)
      print *
      call blha_configuration_final (cfg)
```

2762

```
      call blha_configuration_init (cfg, var_str ("test"), model, &
         mode=BLHA_MODE_GOSAM)
      call blha_configuration_set (cfg, alphas_power = 0, &
         model_file = var_str ("test.slha"))
      pdg_in(1) = [1]
      pdg_in(2) = [-1]
      pdg_out(1) = [22]
      pdg_out(2) = [22]
      call blha_configuration_append_process (cfg, pdg_in, pdg_out(1:2))
      call blha_configuration_freeze (cfg)
      u = free_unit ()
      open (u, file="test.blha.order", action="write", status="replace")
      call blha_configuration_write (cfg, u)
      call blha_configuration_final (cfg)
      inquire (file="test.blha.contract", exist=flag)
      if (.not. flag) return
      call blha_configuration_init (cfg, var_str ("test"), model, mode=BLHA_MODE_GOSAM)
      call blha_read_contract (cfg, ok, var_str ("test.blha.contract"), success=flag)
      print *, "Reading back processed configuration: success? ", ok
   end subroutine blha_config_test
```

21.1.2 OLP matrix element interface

The prototypes of the OLP functions.

⟨*BLHA interface: interfaces*⟩≡

```
   abstract interface
      subroutine ext_olp_start (file, status) bind(c)
        import
        character(c_char), dimension(*), intent(in) :: file
        integer(c_int), intent(out) :: status
      end subroutine ext_olp_Start

      subroutine ext_olp_evalsubprocess &
           (label, momenta, scale, parameters, amp) bind(c)
        import
        integer(c_int), intent(in), value :: label
        real(c_double), dimension(*), intent(in) :: momenta
        real(c_double), intent(in), value :: scale
        real(c_double), dimension(*), intent(in) :: parameters
        real(c_double), dimension(*), intent(out) :: amp
      end subroutine ext_olp_evalsubprocess

      subroutine ext_olp_finalize () bind(c)
      end subroutine ext_olp_finalize

      subroutine ext_olp_option (assignment, status) bind(c)
        import
        character(c_char), dimension(*), intent(in) :: assignment
        integer(c_int), intent(out) :: status
      end subroutine ext_olp_option
   end interface
```

The OLP library is encapsulated together with the configuration in derived type:

⟨*BLHA interface: public*⟩≡
```
public :: blha_olp_t
```

⟨*BLHA interface: types*⟩≡
```
type :: blha_olp_t
   private
   type(blha_configuration_t) :: cfg
   type(string_t) :: library
   integer :: n_in, n_out, n_flv, n_hel, n_col
   integer, dimension(:,:), allocatable :: flv_state
   logical :: color_summed = .true., flavor_summed = .true.
   logical :: loaded = .false.
   type(dlaccess_t) :: lib_handle
   procedure(ext_olp_start), pointer, nopass :: olp_start => null ()
   procedure(ext_olp_evalsubprocess), pointer, nopass :: &
      olp_evalsubprocess => null ()
   procedure(ext_olp_finalize), pointer, nopass :: olp_finalize => null ()
   procedure(ext_olp_option), pointer, nopass :: olp_option => null ()
end type blha_olp_t
```

Init the `blha_olp_t` object and try to dlopen the library.

⟨*BLHA interface: public*⟩+≡
```
public :: blha_olp_init
```

⟨*BLHA interface: procedures*⟩≡
```
subroutine blha_olp_init (olp, cfg, library, success)
  type(blha_olp_t), intent(out) :: olp
  type(string_t), intent(in), optional :: library
  type(blha_configuration_t), intent(in) :: cfg
  logical, intent(out), optional :: success
  type(blha_cfg_process_node_t), pointer :: node
  type(string_t) :: prefix, libname
  type(c_funptr) :: fptr
  integer :: olp_status
  success = .true.
  node => cfg%processes
  if (.not. associated (node)) then
     call error ("blha_interface_init: empty process list")
     return
  end if
  olp%n_in = size (node%pdg_in)
  olp%n_out = size (node%pdg_out)
  do while (associated (node))
     if ((olp%n_in /= size (node%pdg_in)) .or. &
           (olp%n_out /= size (node%pdg_out))) then
        call error ("blha_interface_init: inconsistent process list")
        return
     end if
     node => node%next
  end do
  if (present (library)) then
     olp%library = library
  else
```

```
         olp%library = cfg%name // ".so"
      end if
      if (char (extract (olp%library, 1, 1)) == "/") then
         prefix = ""
         libname = extract (olp%library, 2)
      else
         prefix = "."
         libname = olp%library
      end if
      call dlaccess_init (olp%lib_handle, prefix, libname)
      if (dlaccess_has_error (olp%lib_handle)) then
         call error ("blha_interface_init: error opening library: " // &
            char (dlaccess_get_error (olp%lib_handle)))
         call dlaccess_final (olp%lib_handle)
         return
      end if
      fptr = dlaccess_get_c_funptr (olp%lib_handle, var_str ("OLP_Start"))
      if (.not. check_dlstate ()) return
      call c_f_procpointer (fptr, olp%olp_start)
      fptr = dlaccess_get_c_funptr (olp%lib_handle, var_str ("OLP_EvalSubProcess"))
      if (.not. check_dlstate ()) return
      call c_f_procpointer (fptr, olp%olp_evalsubprocess)
      if (olp%cfg%mode == BLHA_MODE_GOSAM) then
         fptr = dlaccess_get_c_funptr (olp%lib_handle, var_str ("OLP_Finalize"))
         if (.not. check_dlstate ()) return
         call c_f_procpointer (fptr, olp%olp_finalize)
         fptr = dlaccess_get_c_funptr (olp%lib_handle, var_str ("OLP_Option"))
         if (.not. check_dlstate ()) return
         call c_f_procpointer (fptr, olp%olp_option)
      end if
      call olp%olp_start (string_f2c (cfg%model_file), olp_status)
      if (olp_status /= 1) then
         call error ("blha_interface_init: OLP initialization failed")
         call dlaccess_final (olp%lib_handle)
      end if
      success = .true.
      olp%loaded = .true.

   contains

      function check_dlstate () result (ok)
        logical :: ok
        ok = .not. dlaccess_has_error (olp%lib_handle)
        if (.not. ok) then
           call error ("blha_interface_init: error loading library: " // &
               char (dlaccess_get_error (olp%lib_handle)))
           call dlaccess_final (olp%lib_handle)
        end if
      end function check_dlstate

      subroutine error (msg)
        character(*), intent(in) :: msg
        if (present (success)) then
           call msg_error (msg)
```

```
            success = .false.
         else
            call msg_fatal (msg)
         end if
       end subroutine error

    end subroutine blha_olp_init
```

Finalizer.

⟨*BLHA interface: public*⟩+≡
```
    public :: blha_olp_final
```

⟨*BLHA interface: procedures*⟩+≡
```
    subroutine blha_olp_final (olp)
      type(blha_olp_t), intent(inout) :: olp
      if (.not. olp%loaded) return
      if (associated (olp%olp_finalize)) call olp%olp_finalize
      call dlaccess_final (olp%lib_handle)
      olp%loaded = .false.
    end subroutine blha_olp_final
```

Test.

⟨*BLHA interface: public*⟩+≡
```
    public :: blha_interface_test
```

⟨*BLHA interface: procedures*⟩+≡
```
    subroutine blha_interface_test (cfg, ok)
      type(blha_configuration_t), intent(inout) :: cfg
      type(blha_olp_t) :: olp
      logical, intent(out) :: ok
      call blha_olp_init (olp, cfg, library=var_str ("blha_test.so"), success=ok)
      print *, "loading OLP library: success?", ok
      call blha_olp_final (olp)
    end subroutine blha_interface_test
```

21.1.3 OLP driver

⟨*BLHA driver: public*⟩≡
```
    public :: blha_test
```

⟨*BLHA driver: procedures*⟩≡
```
    subroutine blha_test (model)
      type(model_t), pointer :: model
      type (blha_configuration_t) :: cfg
      logical :: ok
      call blha_config_test (model, cfg, ok)
      if (ok) call blha_interface_test (cfg, ok)
    end subroutine blha_test
```

Chapter 22

Integration and simulation

This layer of modules is just below the top-level API. We lay out specific data types for integration and simulation and implement the corresponding algorithms as methods acting on them. This helps to keep the command-level implementation concise and simple.

22.1 Iterations

This module defines a container for the list of iterations and calls, to be submitted to integration.

⟨`iterations.f90`⟩≡
  ⟨*File header*⟩

  ```
  module iterations
  ```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
  ```
    use diagnostics !NODEP!
    use unit_tests
    use processes
  ```

  ⟨*Standard module head*⟩

  ⟨*Iterations: public*⟩

  ⟨*Iterations: types*⟩

  ```
  contains
  ```

  ⟨*Iterations: procedures*⟩

  ⟨*Iterations: tests*⟩

  ```
  end module iterations
  ```

22.1.1 The iterations list

Each integration pass has a number of iterations and a number of calls per iteration. The last pass produces the end result; the previous passes are used for adaptation.

The flags `adapt_grid` and `adapt_weight` are used only if `custom_adaptation` is set. Otherwise, default settings are used that depend on the integration pass.

⟨*Iterations: types*⟩≡
```
type :: iterations_spec_t
  private
  integer :: n_it = 0
  integer :: n_calls = 0
  logical :: custom_adaptation = .false.
  logical :: adapt_grids = .false.
  logical :: adapt_weights = .false.
end type iterations_spec_t
```

We build up a list of iterations.

⟨*Iterations: public*⟩≡
```
public :: iterations_list_t
```

⟨*Iterations: types*⟩+≡
```
type :: iterations_list_t
  private
  integer :: n_pass = 0
  type(iterations_spec_t), dimension(:), allocatable :: pass
 contains
 ⟨Iterations: iterations list: TBP⟩
end type iterations_list_t
```

Initialize an iterations list. For each pass, we have to specify the number of iterations and calls. We may provide the adaption conventions explicitly, either as character codes or as logicals.

For passes where the adaptation conventions are not specified, we use the following default setting: adapt weights and grids for all passes except the last one.

⟨*Iterations: iterations list: TBP*⟩≡
```
procedure :: init => iterations_list_init
```

⟨*Iterations: procedures*⟩≡
```
subroutine iterations_list_init &
    (it_list, n_it, n_calls, adapt, adapt_code, adapt_grids, adapt_weights)
  class(iterations_list_t), intent(inout) :: it_list
  integer, dimension(:), intent(in) :: n_it, n_calls
  logical, dimension(:), intent(in), optional :: adapt
  type(string_t), dimension(:), intent(in), optional :: adapt_code
  logical, dimension(:), intent(in), optional :: adapt_grids, adapt_weights
  integer :: i
  it_list%n_pass = size (n_it)
  if (allocated (it_list%pass)) deallocate (it_list%pass)
  allocate (it_list%pass (it_list%n_pass))
  it_list%pass%n_it = n_it
  it_list%pass%n_calls = n_calls
```

```
        if (present (adapt)) then
           it_list%pass%custom_adaptation = adapt
           do i = 1, it_list%n_pass
              if (adapt(i)) then
                 if (verify (adapt_code(i), "wg") /= 0) then
                    call msg_error ("iteration specification: " &
                         // "adaptation code letters must be 'w' or 'g'")
                 end if
                 it_list%pass(i)%adapt_grids = scan (adapt_code(i), "g") /= 0
                 it_list%pass(i)%adapt_weights = scan (adapt_code(i), "w") /= 0
              end if
           end do
        else if (present (adapt_grids) .and. present (adapt_weights)) then
           it_list%pass%custom_adaptation = .true.
           it_list%pass%adapt_grids = adapt_grids
           it_list%pass%adapt_weights = adapt_weights
        end if
        do i = 1, it_list%n_pass - 1
           if (.not. it_list%pass(i)%custom_adaptation) then
              it_list%pass(i)%adapt_grids = .true.
              it_list%pass(i)%adapt_weights = .true.
           end if
        end do
     end subroutine iterations_list_init
```

⟨*Iterations: iterations list: TBP*⟩+≡
```
     procedure :: clear => iterations_list_clear
```

⟨*Iterations: procedures*⟩+≡
```
     subroutine iterations_list_clear (it_list)
       class(iterations_list_t), intent(inout) :: it_list
       it_list%n_pass = 0
       deallocate (it_list%pass)
     end subroutine iterations_list_clear
```

Write the list of iterations.

⟨*Iterations: iterations list: TBP*⟩+≡
```
     procedure :: write => iterations_list_write
```

⟨*Iterations: procedures*⟩+≡
```
     subroutine iterations_list_write (it_list, unit)
       class(iterations_list_t), intent(in) :: it_list
       integer, intent(in), optional :: unit
       integer :: u
       u = output_unit (unit)
       write (u, "(A)")  char (it_list%to_string ())
     end subroutine iterations_list_write
```

The output as a single-line string.

⟨*Iterations: iterations list: TBP*⟩+≡
```
     procedure :: to_string => iterations_list_to_string
```

⟨*Iterations: procedures*⟩+≡
```
     function iterations_list_to_string (it_list) result (buffer)
```

```
      class(iterations_list_t), intent(in) :: it_list
      type(string_t) :: buffer
      character(30) :: ibuf
      integer :: i
      buffer = "iterations = "
      if (it_list%n_pass > 0) then
         do i = 1, it_list%n_pass
            if (i > 1)  buffer = buffer // ", "
            write (ibuf, "(I0,':',I0)") &
                 it_list%pass(i)%n_it, it_list%pass(i)%n_calls
            buffer = buffer // trim (ibuf)
            if (it_list%pass(i)%custom_adaptation &
                 .or. it_list%pass(i)%adapt_grids &
                 .or. it_list%pass(i)%adapt_weights) then
               buffer = buffer // ':"'
               if (it_list%pass(i)%adapt_grids)  buffer = buffer // "g"
               if (it_list%pass(i)%adapt_weights)  buffer = buffer // "w"
               buffer = buffer // '"'
            end if
         end do
      else
         buffer = buffer // "[undefined]"
      end if
   end function iterations_list_to_string
```

22.1.2 Tools

Return the total number of passes.

⟨*Iterations: iterations list: TBP*⟩+≡
```
   procedure :: get_n_pass => iterations_list_get_n_pass
```

⟨*Iterations: procedures*⟩+≡
```
   function iterations_list_get_n_pass (it_list) result (n_pass)
     class(iterations_list_t), intent(in) :: it_list
     integer :: n_pass
     n_pass = it_list%n_pass
   end function iterations_list_get_n_pass
```

Return the number of calls for a specific pass.

⟨*Iterations: iterations list: TBP*⟩+≡
```
   procedure :: get_n_calls => iterations_list_get_n_calls
```

⟨*Iterations: procedures*⟩+≡
```
   function iterations_list_get_n_calls (it_list, pass) result (n_calls)
     class(iterations_list_t), intent(in) :: it_list
     integer :: n_calls
     integer, intent(in) :: pass
     if (pass <= it_list%n_pass) then
        n_calls = it_list%pass(pass)%n_calls
     else
        n_calls = 0
     end if
   end function iterations_list_get_n_calls
```

Get the adaptation mode (automatic/custom) and, for custom adaptation, the flags for a specific pass.

⟨*Iterations: iterations list: TBP*⟩+≡
```
    procedure :: adapt_grids => iterations_list_adapt_grids
    procedure :: adapt_weights => iterations_list_adapt_weights
```

⟨*Iterations: procedures*⟩+≡
```
    function iterations_list_adapt_grids (it_list, pass) result (flag)
      logical :: flag
      class(iterations_list_t), intent(in) :: it_list
      integer, intent(in) :: pass
      if (pass <= it_list%n_pass) then
         flag = it_list%pass(pass)%adapt_grids
      else
         flag = .false.
      end if
    end function iterations_list_adapt_grids

    function iterations_list_adapt_weights (it_list, pass) result (flag)
      logical :: flag
      class(iterations_list_t), intent(in) :: it_list
      integer, intent(in) :: pass
      if (pass <= it_list%n_pass) then
         flag = it_list%pass(pass)%adapt_weights
      else
         flag = .false.
      end if
    end function iterations_list_adapt_weights
```

Return the total number of iterations / the iterations for a specific pass.

⟨*Iterations: iterations list: TBP*⟩+≡
```
    procedure :: get_n_it => iterations_list_get_n_it
```

⟨*Iterations: procedures*⟩+≡
```
    function iterations_list_get_n_it (it_list, pass) result (n_it)
      class(iterations_list_t), intent(in) :: it_list
      integer :: n_it
      integer, intent(in) :: pass
      if (pass <= it_list%n_pass) then
         n_it = it_list%pass(pass)%n_it
      else
         n_it = 0
      end if
    end function iterations_list_get_n_it
```

22.1.3 Test

This is the master for calling self-test procedures.

⟨*Iterations: public*⟩+≡
```
    public :: iterations_test
```

⟨*Iterations: tests*⟩≡
```
    subroutine iterations_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Iterations: execute tests⟩
  end subroutine iterations_test
```

Empty list

⟨*Iterations: execute tests*⟩≡
```
    call test (iterations_1, "iterations_1", &
        "empty iterations list", &
        u, results)
```
⟨*Iterations: tests*⟩+≡
```
    subroutine iterations_1 (u)
      integer, intent(in) :: u
      type(iterations_list_t) :: it_list

      write (u, "(A)")  "* Test output: iterations_1"
      write (u, "(A)")  "*   Purpose: display empty iterations list"
      write (u, "(A)")

      call it_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: iterations_1"

    end subroutine iterations_1
```

Fill list

⟨*Iterations: execute tests*⟩+≡
```
    call test (iterations_2, "iterations_2", &
        "create iterations list", &
        u, results)
```
⟨*Iterations: tests*⟩+≡
```
    subroutine iterations_2 (u)
      integer, intent(in) :: u
      type(iterations_list_t) :: it_list

      write (u, "(A)")  "* Test output: iterations_2"
      write (u, "(A)")  "*   Purpose: fill and display iterations list"
      write (u, "(A)")

      write (u, "(A)")  "* Minimal setup (2 passes)"
      write (u, "(A)")

      call it_list%init ([2, 4], [5000, 20000])

      call it_list%write (u)
      call it_list%clear ()
```

2772

```
      write (u, "(A)")
      write (u, "(A)")   "* Setup with flags (3 passes)"
      write (u, "(A)")

      call it_list%init ([2, 4, 5], [5000, 20000, 400], &
           [.false., .true., .true.], &
           [var_str (""), var_str ("g"), var_str ("wg")])

      call it_list%write (u)

      write (u, "(A)")
      write (u, "(A)")   "* Extract data"
      write (u, "(A)")

      write (u, "(A,I0)")   "n_pass = ", it_list%get_n_pass ()
      write (u, "(A)")
      write (u, "(A,I0)")   "n_calls(2) = ", it_list%get_n_calls (2)
      write (u, "(A)")
      write (u, "(A,I0)")   "n_it(3) = ", it_list%get_n_it (3)

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: iterations_2"

    end subroutine iterations_2
```

22.2   User-controlled File I/O

The SINDARIN language includes commands that write output to file (input
may be added later). We identify files by their name, and manage the unit
internally. We need procedures for opening, closing, and printing files.

⟨user_files.f90⟩≡
  ⟨*File header*⟩

```
  module user_files
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use ifiles
    use analysis
```

  ⟨*Standard module head*⟩

  ⟨*User files: public*⟩

  ⟨*User files: types*⟩

  ⟨*User files: interfaces*⟩

```
  contains
```

2773

⟨*User files: procedures*⟩

```
  end module user_files
```

22.2.1 The file type

This is a type that describes an open user file and its properties. The entry is part of a doubly-linked list.

⟨*User files: types*⟩≡
```
    type :: file_t
       private
       type(string_t) :: name
       integer :: unit = -1
       logical :: reading = .false.
       logical :: writing = .false.
       type(file_t), pointer :: prev => null ()
       type(file_t), pointer :: next => null ()
    end type file_t
```

The initializer opens the file.

⟨*User files: procedures*⟩≡
```
    subroutine file_init (file, name, action, status, position)
      type(file_t), intent(out) :: file
      type(string_t), intent(in) :: name
      character(len=*), intent(in) :: action, status, position
      file%unit = free_unit ()
      file%name = name
      open (unit = file%unit, file = char (file%name), &
            action = action, status = status, position = position)
      select case (action)
      case ("read")
         file%reading = .true.
      case ("write")
         file%writing = .true.
      case ("readwrite")
         file%reading = .true.
         file%writing = .true.
      end select
    end subroutine file_init
```

The finalizer closes it.

⟨*User files: procedures*⟩+≡
```
    subroutine file_final (file)
      type(file_t), intent(inout) :: file
      close (unit = file%unit)
      file%unit = -1
    end subroutine file_final
```

Check if a file is open with correct status.

⟨*User files: procedures*⟩+≡

```
function file_is_open (file, action) result (flag)
  logical :: flag
  type(file_t), intent(in) :: file
  character(*), intent(in) :: action
  select case (action)
  case ("read")
     flag = file%reading
  case ("write")
     flag = file%writing
  case ("readwrite")
     flag = file%reading .and. file%writing
  case default
     call msg_bug ("Checking file '" // char (file%name) &
           // "': illegal action specifier")
  end select
end function file_is_open
```

Write to the file. Error if in wrong mode. If there is no string, just write an empty record. If there is a string, respect the `advancing` option.

⟨*User files: procedures*⟩+≡
```
subroutine file_write_string (file, string, advancing)
  type(file_t), intent(in) :: file
  type(string_t), intent(in), optional :: string
  logical, intent(in), optional :: advancing
  if (file%writing) then
     if (present (string)) then
        if (present (advancing)) then
           if (advancing) then
              write (file%unit, "(A)")  char (string)
           else
              write (file%unit, "(A)", advance="no")  char (string)
           end if
        else
           write (file%unit, "(A)")  char (string)
        end if
     else
        write (file%unit, *)
     end if
  else
     call msg_error ("Writing to file: File '" // char (file%name) &
           // "' is not open for writing.")
  end if
end subroutine file_write_string
```

Write a whole ifile, line by line.

⟨*User files: procedures*⟩+≡
```
subroutine file_write_ifile (file, ifile)
  type(file_t), intent(in) :: file
  type(ifile_t), intent(in) :: ifile
  type(line_p) :: line
  call line_init (line, ifile)
  do while (line_is_associated (line))
     call file_write_string (file, line_get_string_advance (line))
```

```
        end do
    end subroutine file_write_ifile
```

Write an analysis object (or all objects) to an open file.

⟨*User files: procedures*⟩+≡
```
    subroutine file_write_analysis (file, tag)
      type(file_t), intent(in) :: file
      type(string_t), intent(in), optional :: tag
      if (file%writing) then
         if (present (tag)) then
            call analysis_write (tag, unit = file%unit)
         else
            call analysis_write (unit = file%unit)
         end if
      else
         call msg_error ("Writing analysis to file: File '" // char (file%name) &
              // "' is not open for writing.")
      end if
    end subroutine file_write_analysis
```

22.2.2 The file list

We maintain a list of all open files and their attributes. The list must be doubly-linked because we may delete entries.

⟨*User files: public*⟩≡
```
    public :: file_list_t
```

⟨*User files: types*⟩+≡
```
    type :: file_list_t
      type(file_t), pointer :: first => null ()
      type(file_t), pointer :: last => null ()
    end type file_list_t
```

There is no initialization routine, but a finalizer which deletes all:

⟨*User files: public*⟩+≡
```
    public :: file_list_final
```

⟨*User files: procedures*⟩+≡
```
    subroutine file_list_final (file_list)
      type(file_list_t), intent(inout) :: file_list
      type(file_t), pointer :: current
      do while (associated (file_list%first))
         current => file_list%first
         file_list%first => current%next
         call file_final (current)
         deallocate (current)
      end do
      file_list%last => null ()
    end subroutine file_list_final
```

Find an entry in the list. Return null pointer on failure.

⟨*User files: procedures*⟩+≡
```
function file_list_get_file_ptr (file_list, name) result (current)
  type(file_t), pointer :: current
  type(file_list_t), intent(in) :: file_list
  type(string_t), intent(in) :: name
  current => file_list%first
  do while (associated (current))
     if (current%name == name)  return
     current => current%next
  end do
end function file_list_get_file_ptr
```

Check if a file is open, public version:

⟨*User files: public*⟩+≡
```
public :: file_list_is_open
```

⟨*User files: procedures*⟩+≡
```
function file_list_is_open (file_list, name, action) result (flag)
  logical :: flag
  type(file_list_t), intent(in) :: file_list
  type(string_t), intent(in) :: name
  character(len=*), intent(in) :: action
  type(file_t), pointer :: current
  current => file_list_get_file_ptr (file_list, name)
  if (associated (current)) then
     flag = file_is_open (current, action)
  else
     flag = .false.
  end if
end function file_list_is_open
```

Append a new file entry, i.e., open this file. Error if it is already open.

⟨*User files: public*⟩+≡
```
public :: file_list_open
```

⟨*User files: procedures*⟩+≡
```
subroutine file_list_open (file_list, name, action, status, position)
  type(file_list_t), intent(inout) :: file_list
  type(string_t), intent(in) :: name
  character(len=*), intent(in) :: action, status, position
  type(file_t), pointer :: current
  if (.not. associated (file_list_get_file_ptr (file_list, name))) then
     allocate (current)
     call msg_message ("Opening file '" // char (name) // "' for output")
     call file_init (current, name, action, status, position)
     if (associated (file_list%last)) then
        file_list%last%next => current
        current%prev => file_list%last
     else
        file_list%first => current
     end if
     file_list%last => current
  else
```

2777

```
          call msg_error ("Opening file: File '" // char (name) &
               // "' is already open.")
        end if
      end subroutine file_list_open
```

Delete a file entry, i.e., close this file. Error if it is not open.

⟨User files: public⟩+≡
```
    public :: file_list_close
```

⟨User files: procedures⟩+≡
```
    subroutine file_list_close (file_list, name)
      type(file_list_t), intent(inout) :: file_list
      type(string_t), intent(in) :: name
      type(file_t), pointer :: current
      current => file_list_get_file_ptr (file_list, name)
      if (associated (current)) then
         if (associated (current%prev)) then
            current%prev%next => current%next
         else
            file_list%first => current%next
         end if
         if (associated (current%next)) then
            current%next%prev => current%prev
         else
            file_list%last => current%prev
         end if
         call msg_message ("Closing file '" // char (name) // "' for output")
         call file_final (current)
         deallocate (current)
      else
         call msg_error ("Closing file: File '" // char (name) &
               // "' is not open.")
      end if
    end subroutine file_list_close
```

Write a string to file. Error if it is not open.

⟨User files: public⟩+≡
```
    public :: file_list_write
```

⟨User files: interfaces⟩≡
```
    interface file_list_write
       module procedure file_list_write_string
       module procedure file_list_write_ifile
    end interface
```

⟨User files: procedures⟩+≡
```
    subroutine file_list_write_string (file_list, name, string, advancing)
      type(file_list_t), intent(in) :: file_list
      type(string_t), intent(in) :: name
      type(string_t), intent(in), optional :: string
      logical, intent(in), optional :: advancing
      type(file_t), pointer :: current
      current => file_list_get_file_ptr (file_list, name)
      if (associated (current)) then
```

```
        call file_write_string (current, string, advancing)
     else
        call msg_error ("Writing to file: File '" // char (name) &
             // "'is not open.")
     end if
   end subroutine file_list_write_string

   subroutine file_list_write_ifile (file_list, name, ifile)
     type(file_list_t), intent(in) :: file_list
     type(string_t), intent(in) :: name
     type(ifile_t), intent(in) :: ifile
     type(line_p) :: line
     type(file_t), pointer :: current
     current => file_list_get_file_ptr (file_list, name)
     if (associated (current)) then
        call file_write_ifile (current, ifile)
     else
        call msg_error ("Writing to file: File '" // char (name) &
             // "'is not open.")
     end if
   end subroutine file_list_write_ifile
```

Write an analysis object or all objects to data file. Error if it is not open. If the file name is empty, write to standard output.

⟨*User files: public*⟩+≡
```
    public :: file_list_write_analysis
```

⟨*User files: procedures*⟩+≡
```
    subroutine file_list_write_analysis (file_list, name, tag)
      type(file_list_t), intent(in) :: file_list
      type(string_t), intent(in) :: name
      type(string_t), intent(in), optional :: tag
      type(file_t), pointer :: current
      if (name == "") then
         if (present (tag)) then
            call analysis_write (tag)
         else
            call analysis_write
         end if
      else
         current => file_list_get_file_ptr (file_list, name)
         if (associated (current)) then
            call file_write_analysis (current, tag)
         else
            call msg_error ("Writing analysis to file: File '" // char (name) &
                 // "' is not open.")
         end if
      end if
    end subroutine file_list_write_analysis
```

2779

22.3 Runtime data

⟨rt_data.f90⟩≡
  ⟨*File header*⟩

  module rt_data

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use system_dependencies !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use pdf_builtin !NODEP!
    use sf_lhapdf !NODEP!
    use os_interface
    use ifiles
    use lexers
    use parser
    use models
    use flavors
    use jets
    use variables
    use expressions
    use polarizations
    use beams
    use process_libraries
    use prclib_stacks
    use prc_core
    use beam_structures
    use user_files
    use process_stacks
    use iterations

  ⟨*Standard module head*⟩

  ⟨*RT data: public*⟩

  ⟨*RT data: types*⟩

  contains

  ⟨*RT data: procedures*⟩

  ⟨*RT data: tests*⟩

  end module rt_data

22.3.1 Container for parse nodes

The runtime data set contains a bunch of parse nodes (chunks of code that have
not been compiled into evaluation trees but saved for later use). We collect
them here.

This implementation has the useful effect that an assignment between two objects of this type will establish a pointer-target relationship for all components.

⟨*RT data: types*⟩≡
```
type :: rt_parse_nodes_t
   type(parse_node_t), pointer :: cuts_lexpr => null ()
   type(parse_node_t), pointer :: scale_expr => null ()
   type(parse_node_t), pointer :: fac_scale_expr => null ()
   type(parse_node_t), pointer :: ren_scale_expr => null ()
   type(parse_node_t), pointer :: weight_expr => null ()
   type(parse_node_t), pointer :: selection_lexpr => null ()
   type(parse_node_t), pointer :: reweight_expr => null ()
   type(parse_node_t), pointer :: analysis_lexpr => null ()
   type(parse_node_p), dimension(:), allocatable :: alt_setup
 contains
 ⟨RT data: rt parse nodes: TBP⟩
 end type rt_parse_nodes_t
```

Clear individual components. The parse nodes are nullified. No finalization needed since the pointer targets are part of the global parse tree.

⟨*RT data: rt parse nodes: TBP*⟩≡
```
procedure :: clear => rt_parse_nodes_clear
```

⟨*RT data: procedures*⟩≡
```
subroutine rt_parse_nodes_clear (rt_pn, name)
  class(rt_parse_nodes_t), intent(inout) :: rt_pn
  type(string_t), intent(in) :: name
  select case (char (name))
  case ("cuts")
     rt_pn%cuts_lexpr => null ()
  case ("scale")
     rt_pn%scale_expr => null ()
  case ("factorization_scale")
     rt_pn%fac_scale_expr => null ()
  case ("renormalization_scale")
     rt_pn%ren_scale_expr => null ()
  case ("weight")
     rt_pn%weight_expr => null ()
  case ("selection")
     rt_pn%selection_lexpr => null ()
  case ("reweight")
     rt_pn%reweight_expr => null ()
  case ("analysis")
     rt_pn%analysis_lexpr => null ()
  end select
end subroutine rt_parse_nodes_clear
```

Output for the parse nodes.

⟨*RT data: rt parse nodes: TBP*⟩+≡
```
procedure :: write => rt_parse_nodes_write
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_parse_nodes_write (object, unit)
```

```
      class(rt_parse_nodes_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      call wrt ("Cuts", object%cuts_lexpr)
      call write_separator (u)
      call wrt ("Scale", object%scale_expr)
      call write_separator (u)
      call wrt ("Factorization scale", object%fac_scale_expr)
      call write_separator (u)
      call wrt ("Renormalization scale", object%ren_scale_expr)
      call write_separator (u)
      call wrt ("Weight", object%weight_expr)
      call write_separator_double (u)
      call wrt ("Event selection", object%selection_lexpr)
      call write_separator (u)
      call wrt ("Event reweighting factor", object%reweight_expr)
      call write_separator (u)
      call wrt ("Event analysis", object%analysis_lexpr)
      if (allocated (object%alt_setup)) then
         call write_separator_double (u)
         write (u, "(1x,A,':')")  "Alternative setups"
         do i = 1, size (object%alt_setup)
            call write_separator (u)
            call wrt ("Commands", object%alt_setup(i)%ptr)
         end do
      end if
    contains
      subroutine wrt (title, pn)
        character(*), intent(in) :: title
        type(parse_node_t), intent(in), pointer :: pn
        if (associated (pn)) then
           write (u, "(1x,A,':')")  title
           call write_separator (u)
           call parse_node_write_rec (pn, u)
        else
           write (u, "(1x,A,':',1x,A)")  title, "[undefined]"
        end if
      end subroutine wrt
    end subroutine rt_parse_nodes_write
```

Screen output for individual components. (This should eventually be more
condensed, currently we print the internal representation tree.)

⟨*RT data: rt parse nodes: TBP*⟩+≡
```
    procedure :: show => rt_parse_nodes_show
```

⟨*RT data: procedures*⟩+≡
```
    subroutine rt_parse_nodes_show (rt_pn, name, unit)
      class(rt_parse_nodes_t), intent(in) :: rt_pn
      type(string_t), intent(in) :: name
      integer, intent(in), optional :: unit
      type(parse_node_t), pointer :: pn
      integer :: u
      u = output_unit (unit)
```

```
          select case (char (name))
          case ("cuts")
             pn => rt_pn%cuts_lexpr
          case ("scale")
             pn => rt_pn%scale_expr
          case ("factorization_scale")
             pn => rt_pn%fac_scale_expr
          case ("renormalization_scale")
             pn => rt_pn%ren_scale_expr
          case ("weight")
             pn => rt_pn%weight_expr
          case ("selection")
             pn => rt_pn%selection_lexpr
          case ("reweight")
             pn => rt_pn%reweight_expr
          case ("analysis")
             pn => rt_pn%analysis_lexpr
          end select
          if (associated (pn)) then
             write (u, "(A,1x,A,1x,A)")  "Expression:", char (name), "(parse tree):"
             call parse_node_write_rec (pn, u)
          else
             write (u, "(A,1x,A,A)")  "Expression:", char (name), ": [undefined]"
          end if
        end subroutine rt_parse_nodes_show
```

22.3.2 Container for particle data

If we locally modify particle properties, we must be able to revert them to their
original status. This transparent container stores the original state for a stack
of particles.

⟨*RT data: types*⟩+≡
```
    type :: rt_particle_entry_t
       integer :: pdg = 0
       logical :: stable = .true.
       logical :: isotropic = .false.
       logical :: diagonal = .false.
       logical :: polarized = .false.
       type(rt_particle_entry_t), pointer :: next => null ()
    end type rt_particle_entry_t
```

⟨*RT data: types*⟩+≡
```
    type :: rt_particle_stack_t
       type(model_t), pointer :: model => null ()
       type(rt_particle_entry_t), pointer :: first => null ()
     contains
     ⟨RT data: rt particle stack: TBP⟩
    end type rt_particle_stack_t
```

Finalize the stack.

⟨*RT data: rt particle stack: TBP*⟩≡

```
      procedure :: final => rt_particle_stack_final
⟨RT data: procedures⟩+≡
    subroutine rt_particle_stack_final (object)
      class(rt_particle_stack_t), intent(inout) :: object
      type(rt_particle_entry_t), pointer :: entry
      do while (associated (object%first))
         entry => object%first
         object%first => entry%next
         deallocate (entry)
      end do
      object%model => null ()
    end subroutine rt_particle_stack_final
```

Write the current contents of the particle stack, if it is nonempty.
⟨*RT data: rt particle stack: TBP*⟩+≡
```
      procedure :: write => rt_particle_stack_write
```
⟨*RT data: procedures*⟩+≡
```
    subroutine rt_particle_stack_write (object, unit)
      class(rt_particle_stack_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      type(rt_particle_entry_t), pointer :: entry
      u = output_unit (unit)
      if (.not. object%is_empty ()) then
         write (u, "(1x,A)")  "Stored particle data"
         if (associated (object%model)) then
            write (u, "(3x,A,A)") "Model = ", char (object%model%get_name ())
         else
            write (u, "(3x,A,A)") "Model = [undefined]"
         end if
         entry => object%first
         do while (associated (entry))
            write (u, "(3x,I0,':',4(2x,A,L1))")  entry%pdg, &
                 "stable = ", entry%stable, &
                 "isotropic = ", entry%isotropic, &
                 "diagonal = ", entry%diagonal, &
                 "polarized = ", entry%polarized
            entry => entry%next
         end do
      end if
    end subroutine rt_particle_stack_write
```

Initialize for a model.
⟨*RT data: rt particle stack: TBP*⟩+≡
```
      procedure :: init => rt_particle_stack_init
```
⟨*RT data: procedures*⟩+≡
```
    subroutine rt_particle_stack_init (stack, model)
      class(rt_particle_stack_t), intent(out) :: stack
      type(model_t), intent(in), target :: model
      stack%model => model
    end subroutine rt_particle_stack_init
```

Reset a stack: if the model changes, discard the previous stack and re-initialize.

⟨*RT data: rt particle stack: TBP*⟩+≡
```
procedure :: reset => rt_particle_stack_reset
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_particle_stack_reset (stack, model)
  class(rt_particle_stack_t), intent(inout) :: stack
  type(model_t), intent(in), target :: model
  if (associated (stack%model)) then
     if (model%get_name () /= stack%model%get_name ()) then
        call stack%final ()
        call stack%init (model)
     end if
  else
     call stack%init (model)
  end if
end subroutine rt_particle_stack_reset
```

Push an entry on the stack, storing the current state of the particle in the model.

⟨*RT data: rt particle stack: TBP*⟩+≡
```
procedure :: push => rt_particle_stack_push
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_particle_stack_push (stack, pdg)
  class(rt_particle_stack_t), intent(inout) :: stack
  integer, intent(in) :: pdg
  type(rt_particle_entry_t), pointer :: entry
  type(particle_data_t), pointer :: prt_data
  logical :: anti
  allocate (entry)
  entry%pdg = pdg
  anti = pdg < 0
  prt_data => model_get_particle_ptr (stack%model, pdg)
  entry%stable = particle_data_is_stable (prt_data, anti)
  entry%polarized = particle_data_is_polarized (prt_data, anti)
  entry%isotropic = particle_data_decays_isotropically (prt_data, anti)
  entry%diagonal = particle_data_decays_diagonal (prt_data, anti)
  entry%next => stack%first
  stack%first => entry
end subroutine rt_particle_stack_push
```

⟨*RT data: rt particle stack: TBP*⟩+≡
```
procedure :: is_empty => rt_particle_stack_is_empty
```

⟨*RT data: procedures*⟩+≡
```
function rt_particle_stack_is_empty (stack) result (flag)
  class(rt_particle_stack_t), intent(in) :: stack
  logical :: flag
  flag = .not. associated (stack%first)
end function rt_particle_stack_is_empty
```

Check whether a particle is already on the stack.

⟨*RT data: rt particle stack: TBP*⟩+≡
```
procedure :: contains => rt_particle_stack_contains
```

⟨*RT data: procedures*⟩+≡
```
function rt_particle_stack_contains (stack, pdg) result (flag)
  class(rt_particle_stack_t), intent(in) :: stack
  integer, intent(in) :: pdg
  logical :: flag
  type(rt_particle_entry_t), pointer :: entry
  flag = .false.
  entry => stack%first
  do while (associated (entry))
     if (entry%pdg == pdg) then
        flag = .true.;  return
     end if
     entry => entry%next
  end do
end function rt_particle_stack_contains
```

Restore the previous state of the model that is referenced by the particle stack.
Then, we can delete the stack since it is no longer necessary.

⟨*RT data: rt particle stack: TBP*⟩+≡
```
procedure :: restore_model => rt_particle_stack_restore_model
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_particle_stack_restore_model (stack)
  class(rt_particle_stack_t), intent(inout) :: stack
  type(rt_particle_entry_t), pointer :: entry
  type(particle_data_t), pointer :: prt_data
  if (associated (stack%model)) then
     entry => stack%first
     do while (associated (entry))
        prt_data => model_get_particle_ptr (stack%model, entry%pdg)
        if (entry%pdg > 0) then
           call particle_data_set (prt_data, &
                p_is_stable = entry%stable, &
                p_polarized = entry%polarized, &
                p_decays_isotropically = entry%isotropic, &
                p_decays_diagonal = entry%diagonal)
        else
           call particle_data_set (prt_data, &
                a_is_stable = entry%stable, &
                a_polarized = entry%polarized, &
                a_decays_isotropically = entry%isotropic, &
                a_decays_diagonal = entry%diagonal)
        end if
        entry => entry%next
     end do
  end if
  call stack%final ()
end subroutine rt_particle_stack_restore_model
```

22.3.3   The data type

This is a big data container which contains everything that is used and modified
during the command flow. A local copy of this can be used to temporarily

override defaults. The data set is transparent.

⟨*RT data: public*⟩≡
```
    public :: rt_data_t
```

⟨*RT data: types*⟩+≡
```
    type :: rt_data_t
       type(lexer_t), pointer :: lexer => null ()
       type(var_list_t) :: var_list
       type(iterations_list_t) :: it_list
       type(os_data_t) :: os_data
       type(model_list_t) :: model_list
       type(model_t), pointer :: model => null ()
       type(rt_particle_stack_t) :: particle_stack
       type(prclib_stack_t) :: prclib_stack
       type(process_library_t), pointer :: prclib => null ()
       type(beam_structure_t) :: beam_structure
       type(pdf_builtin_status_t) :: pdf_builtin_status
       type(lhapdf_status_t) :: lhapdf_status
       type(rt_parse_nodes_t) :: pn
       type(process_stack_t) :: process_stack
       type(string_t), dimension(:), allocatable :: sample_fmt
       type(file_list_t), pointer :: out_files => null ()
       logical :: quit = .false.
       integer :: quit_code = 0
       type(string_t) :: logfile
     contains
     ⟨RT data: rt data: TBP⟩
    end type rt_data_t
```

22.3.4 Output

⟨*RT data: rt data: TBP*⟩≡
```
    procedure :: write => rt_data_write
```

⟨*RT data: procedures*⟩+≡
```
    subroutine rt_data_write (object, unit, vars, pacify)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      type(string_t), dimension(:), intent(in), optional :: vars
      logical, intent(in), optional :: pacify
      integer :: u, i
      u = output_unit (unit)
      call write_separator_double (u)
      write (u, "(1x,A)")  "Runtime data:"
      if (present (vars)) then
         if (size (vars) /= 0) then
            call write_separator_double (u)
            write (u, "(1x,A)")  "Selected variables:"
            call write_separator (u)
            call object%write_vars (u, vars)
         end if
      else
         call write_separator_double (u)
         call var_list_write (object%var_list, u, follow_link=.true.)
```

```
      end if
      if (object%it_list%get_n_pass () > 0) then
         call write_separator_double (u)
         write (u, "(1x)", advance="no")
         call object%it_list%write (u)
      end if
      if (associated (object%model)) then
         call write_separator_double (u)
         call object%model_list%write (u)
         if (.not. object%particle_stack%is_empty ()) then
            call write_separator (u)
            call object%particle_stack%write (u)
         end if
      end if
      call object%prclib_stack%write (u)
      call object%beam_structure%write (u)
      call write_separator_double (u)
      call object%pn%write (u)
      if (allocated (object%sample_fmt)) then
         call write_separator (u)
         write (u, "(1x,A)", advance="no")  "Event sample formats = "
         do i = 1, size (object%sample_fmt)
            if (i > 1)  write (u, "(A,1x)", advance="no")  ","
            write (u, "(A)", advance="no")  char (object%sample_fmt(i))
         end do
         write (u, "(A)")
      end if
      call object%process_stack%write (u, pacify)
      write (u, "(1x,A,1x,L1)")  "quit      :", object%quit
      write (u, "(1x,A,1x,I0)")  "quit_code:", object%quit_code
      call write_separator_double (u)
      write (u, "(1x,A,1x,A)")   "Logfile  :", "'" // trim (char (object%logfile)) // "'"
      call write_separator_double (u)
    end subroutine rt_data_write
```

Write only selected variables.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: write_vars => rt_data_write_vars
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_write_vars (object, unit, vars)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      type(string_t), dimension(:), intent(in), optional :: vars
      integer :: u, i
      u = output_unit (unit)
      if (present (vars)) then
         do i = 1, size (vars)
            call var_list_write_var (object%var_list, vars(i), unit = u, &
                 follow_link = .true.)
         end do
      end if
    end subroutine rt_data_write_vars
```

Write only the model list.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: write_model_list => rt_data_write_model_list
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_write_model_list (object, unit)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      call object%model_list%write (u)
    end subroutine rt_data_write_model_list
```

Write only the library stack.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: write_libraries => rt_data_write_libraries
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_write_libraries (object, unit, libpath)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: libpath
      integer :: u
      u = output_unit (unit)
      call object%prclib_stack%write (u, libpath)
    end subroutine rt_data_write_libraries
```

Write only the beam data.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: write_beams => rt_data_write_beams
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_write_beams (object, unit)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      call write_separator_double (u)
      call object%beam_structure%write (u)
      call write_separator_double (u)
    end subroutine rt_data_write_beams
```

Write only the process and event expressions.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: write_expr => rt_data_write_expr
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_write_expr (object, unit)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      call write_separator_double (u)
      call object%pn%write (u)
```

```
      call write_separator_double (u)
    end subroutine rt_data_write_expr
```

Write only the process stack.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: write_process_stack => rt_data_write_process_stack
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_write_process_stack (object, unit)
      class(rt_data_t), intent(in) :: object
      integer, intent(in), optional :: unit
      call object%process_stack%write (unit)
    end subroutine rt_data_write_process_stack
```

22.3.5 Clear

The `clear` command can remove the contents of various subobjects. The objects themselves should stay.

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: clear_beams => rt_data_clear_beams
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_clear_beams (global)
      class(rt_data_t), intent(inout) :: global
      call global%beam_structure%final_sf ()
      call global%beam_structure%final_pol ()
      call global%beam_structure%final_mom ()
    end subroutine rt_data_clear_beams
```

22.3.6 Initialization

Initialize runtime data. This defines special variables such as `sqrts`, and should be done only for the instance that is actually global. Local copies will inherit the special variables.

   We link the global variable list to the process stack variable list, so the latter is always available (and kept global).

⟨RT data: rt data: TBP⟩+≡
```
    procedure :: global_init => rt_data_global_init
```

⟨RT data: procedures⟩+≡
```
    subroutine rt_data_global_init (global, paths, logfile)
      class(rt_data_t), intent(out), target :: global
      type(paths_t), intent(in), optional :: paths
      type(string_t), intent(in), optional :: logfile
      logical, target, save :: known = .true.
      integer :: seed
      real(default), parameter :: real_specimen = 1.
      call os_data_init (global%os_data, paths)
      if (present (logfile)) then
         global%logfile = logfile
      else
```

```
      global%logfile = ""
   end if
   allocate (global%out_files)
   call system_clock (seed)
   call var_list_append_log_ptr &
        (global%var_list, var_str ("?logging"), logging, known, &
        intrinsic=.true.)
   call var_list_append_int &
        (global%var_list, var_str ("seed"), seed, &
         intrinsic=.true.)
   call var_list_append_string &
        (global%var_list, var_str ("$model_name"), &
         intrinsic=.true.)
   call var_list_append_int &
        (global%var_list, var_str ("process_num_id"), &
        intrinsic=.true.)
   call var_list_append_string &
        (global%var_list, var_str ("$method"), var_str ("omega"), &
        intrinsic=.true.)
   call var_list_append_log &
        (global%var_list, var_str ("?report_progress"), .true., &
         intrinsic=.true.)
   call var_list_append_string &
        (global%var_list, var_str ("$restrictions"), var_str (""), &
        intrinsic=.true.)
   call var_list_append_string &
        (global%var_list, var_str ("$omega_flags"), var_str (""), &
        intrinsic=.true.)
   call var_list_append_log &
        (global%var_list, var_str ("?read_color_factors"), .true., &
         intrinsic=.true.)
!!! JRR: WK please check (#529)
!    call var_list_append_string &
!         (global%var_list, var_str ("$user_procs_cut"), var_str (""), &
!          intrinsic=.true.)
!    call var_list_append_string &
!         (global%var_list, var_str ("$user_procs_event_shape"), var_str (""), &
!          intrinsic=.true.)
!    call var_list_append_string &
!         (global%var_list, var_str ("$user_procs_obs1"), var_str (""), &
!          intrinsic=.true.)
!    call var_list_append_string &
!         (global%var_list, var_str ("$user_procs_obs2"), var_str (""), &
!          intrinsic=.true.)
!    call var_list_append_string &
!         (global%var_list, var_str ("$user_procs_sf"), var_str (""), &
!          intrinsic=.true.)
   call var_list_append_log &
        (global%var_list, var_str ("?slha_read_input"), .true., &
         intrinsic=.true.)
   call var_list_append_log &
        (global%var_list, var_str ("?slha_read_spectrum"), .true., &
         intrinsic=.true.)
   call var_list_append_log &
```

```
          (global%var_list, var_str ("?slha_read_decays"), .false., &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$library_name"), &
           intrinsic=.true.)
     call var_list_append_real &
          (global%var_list, var_str ("sqrts"), &
           intrinsic=.true.)
     call var_list_append_real &
          (global%var_list, var_str ("luminosity"), 0._default, &
           intrinsic=.true.)
     call var_list_append_log &
          (global%var_list, var_str ("?sf_trace"), .false., &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$sf_trace_file"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_log &
          (global%var_list, var_str ("?sf_allow_s_mapping"), .true., &
           intrinsic=.true.)
     if (present (paths)) then
        call var_list_append_string &
             (global%var_list, var_str ("$lhapdf_dir"), paths%lhapdfdir, &
              intrinsic=.true.)
     else
        call var_list_append_string &
             (global%var_list, var_str ("$lhapdf_dir"), var_str(""), &
              intrinsic=.true.)
     end if
     call var_list_append_string &
          (global%var_list, var_str ("$lhapdf_file"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_string &
          (global%var_list, var_str ("$lhapdf_photon_file"), var_str (""), &
           intrinsic=.true.)
     call var_list_append_int &
          (global%var_list, var_str ("lhapdf_member"), 0, &
           intrinsic=.true.)
     call var_list_append_int &
          (global%var_list, var_str ("lhapdf_photon_scheme"), 0, &
           intrinsic=.true.)
     call var_list_append_log &
          (global%var_list, var_str ("?lhapdf_hoppet_b_matching"), .false., &
           intrinsic=.true.)
     call var_list_append_real &
          (global%var_list, var_str ("isr_alpha"), 0._default, &
           intrinsic=.true.)
     call var_list_append_real &
          (global%var_list, var_str ("isr_q_max"), 0._default, &
           intrinsic=.true.)
     call var_list_append_real &
          (global%var_list, var_str ("isr_mass"), 0._default, &
           intrinsic=.true.)
     call var_list_append_int &
```

```
          (global%var_list, var_str ("isr_order"), 3, &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?isr_recoil"), .false., &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("epa_alpha"), 0._default, &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("epa_x_min"), 0._default, &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("epa_q_min"), 0._default, &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("epa_e_max"), 0._default, &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("epa_mass"), 0._default, &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?epa_recoil"), .false., &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("ewa_x_min"), 0._default, &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("ewa_pt_max"), 0._default, &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("ewa_mass"), 0._default, &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?ewa_keep_momentum"), .false., &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?ewa_keep_energy"), .false., &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?circe1_photon1"), .false., &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?circe1_photon2"), .false., &
           intrinsic=.true.)
    call var_list_append_real &
          (global%var_list, var_str ("circe1_sqrts"), &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?circe1_generate"), .true., &
           intrinsic=.true.)
    call var_list_append_log &
          (global%var_list, var_str ("?circe1_map"), .true., &
           intrinsic=.true.)
    call var_list_append_real &
```

```
       (global%var_list, var_str ("circe1_mapping_slope"), 2._default, &
        intrinsic=.true.)
  call var_list_append_real &
       (global%var_list, var_str ("circe1_eps"), 1e-5_default, &
        intrinsic=.true.)
  call var_list_append_int &
       (global%var_list, var_str ("circe1_ver"), 0, intrinsic=.true.)
  call var_list_append_int &
       (global%var_list, var_str ("circe1_rev"), 0, intrinsic=.true.)
  call var_list_append_string &
       (global%var_list, var_str ("$circe1_acc"), var_str ("SBAND"), &
        intrinsic=.true.)
  call var_list_append_int &
       (global%var_list, var_str ("circe1_chat"), 0, intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?circe2_polarized"), .true., &
        intrinsic=.true.)
  call var_list_append_string &
       (global%var_list, var_str ("$circe2_file"), &
        intrinsic=.true.)
  call var_list_append_string &
       (global%var_list, var_str ("$circe2_design"), var_str ("*"), &
        intrinsic=.true.)
  call var_list_append_string &
       (global%var_list, var_str ("$beam_events_file"), &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?beam_events_warn_eof"), .true., &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?energy_scan_normalize"), .false., &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?alpha_s_is_fixed"), .true., &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?alpha_s_from_lhapdf"), .false., &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?alpha_s_from_pdf_builtin"), .false., &
        intrinsic=.true.)
  call var_list_append_int &
       (global%var_list, var_str ("alpha_s_order"), 0, &
        intrinsic=.true.)
  call var_list_append_int &
       (global%var_list, var_str ("alpha_s_nf"), 5, &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?alpha_s_from_mz"), .false., &
        intrinsic=.true.)
  call var_list_append_log &
       (global%var_list, var_str ("?alpha_s_from_lambda_qcd"), .false., &
        intrinsic=.true.)
  call var_list_append_real &
```

```
             (global%var_list, var_str ("lambda_qcd"), 200.e-3_default, &
              intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?fatal_beam_decay"), .true., &
              intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?helicity_selection_active"), .true., &
              intrinsic=.true.)
         call var_list_append_real &
             (global%var_list, var_str ("helicity_selection_threshold"), &
              1E10_default, &
              intrinsic=.true.)
         call var_list_append_int &
             (global%var_list, var_str ("helicity_selection_cutoff"), 1000, &
              intrinsic=.true.)
         call var_list_append_string &
             (global%var_list, var_str ("$rng_method"), var_str ("tao"), &
              intrinsic=.true.)
         call var_list_append_string &
             (global%var_list, var_str ("$integration_method"), var_str ("vamp"), &
              intrinsic=.true.)
         call var_list_append_int &
             (global%var_list, var_str ("threshold_calls"), 10, &
              intrinsic=.true.)
         call var_list_append_int &
             (global%var_list, var_str ("min_calls_per_channel"), 10, &
              intrinsic=.true.)
         call var_list_append_int &
             (global%var_list, var_str ("min_calls_per_bin"), 10, &
              intrinsic=.true.)
         call var_list_append_int &
             (global%var_list, var_str ("min_bins"), 3, &
              intrinsic=.true.)
         call var_list_append_int &
             (global%var_list, var_str ("max_bins"), 20, &
              intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?stratified"), .true., &
              intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?use_vamp_equivalences"), .true., &
              intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?vamp_verbose"), .false., &
             intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?vamp_history_global"), &
             .true., intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?vamp_history_global_verbose"), &
             .false., intrinsic=.true.)
         call var_list_append_log &
             (global%var_list, var_str ("?vamp_history_channels"), &
             .false., intrinsic=.true.)
```

```
call var_list_append_log &
     (global%var_list, var_str ("?vamp_history_channels_verbose"), &
     .false., intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("channel_weights_power"), 0.25_default, &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$phs_method"), var_str ("default"), &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?vis_channels"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?check_phs_file"), .true., &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$phs_file"), var_str (""), &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?phs_only"), .false., &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_threshold_s"), 50._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_threshold_t"), 100._default, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("phs_off_shell"), 2, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("phs_t_channel"), 6, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_e_scale"), 10._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_m_scale"), 10._default, &
      intrinsic=.true.)
call var_list_append_real &
     (global%var_list, var_str ("phs_q_scale"), 10._default, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?phs_keep_nonresonant"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?phs_step_mapping"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?phs_step_mapping_exp"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?phs_s_mapping"), .true., &
      intrinsic=.true.)
```

```
            call var_list_append_string &
                 (global%var_list, var_str ("$run_id"), var_str (""), &
                  intrinsic=.true.)
            call var_list_append_int &
                 (global%var_list, var_str ("n_calls_test"), 0, &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?integration_timer"), .true., &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?check_grid_file"), .true., &
                  intrinsic=.true.)
            call var_list_append_real &
                 (global%var_list, var_str ("accuracy_goal"), 0._default, &
                  intrinsic=.true.)
            call var_list_append_real &
                 (global%var_list, var_str ("error_goal"), 0._default, &
                  intrinsic=.true.)
            call var_list_append_real &
                 (global%var_list, var_str ("relative_error_goal"), 0._default, &
                  intrinsic=.true.)
            call var_list_append_real &
                 (global%var_list, var_str ("error_threshold"), &
                 0._default, intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?vis_history"), .true., &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?diags"), .false., &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?diags_color"), .false., &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?check_event_file"), .true., &
                  intrinsic=.true.)
            call var_list_append_string &
                 (global%var_list, var_str ("$event_file_version"), var_str (""), &
                  intrinsic=.true.)
            call var_list_append_int &
                 (global%var_list, var_str ("n_events"), 0, &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?unweighted"), .true., &
                  intrinsic=.true.)
            call var_list_append_real &
                 (global%var_list, var_str ("safety_factor"), 1._default, &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?negative_weights"), .false., &
                  intrinsic=.true.)
            call var_list_append_log &
                 (global%var_list, var_str ("?keep_beams"), .false., &
                  intrinsic=.true.)
```

```
call var_list_append_log &
     (global%var_list, var_str ("?recover_beams"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?update_event"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?update_sqme"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?update_weight"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?allow_decays"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?auto_decays"), .false., &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("auto_decays_multiplicity"), 2, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?auto_decays_radiative"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?decay_rest_frame"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?isotropic_decay"), .false., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?diagonal_decay"), .false., &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$sample"), var_str (""), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$sample_normalization"), var_str ("auto"),&
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?sample_pacify"), .false., &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("sample_max_tries"), 10000, &
     intrinsic = .true.)
call var_list_append_int &
     (global%var_list, var_str ("sample_split_n_evt"), 0, &
     intrinsic = .true.)
call var_list_append_int &
     (global%var_list, var_str ("sample_split_index"), 0, &
     intrinsic = .true.)
call var_list_append_string &
     (global%var_list, var_str ("$rescan_input_format"), var_str ("raw"), &
      intrinsic=.true.)
```

```
call var_list_append_log &
     (global%var_list, var_str ("?read_raw"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?write_raw"), .true., &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_raw"), var_str ("evx"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_default"), var_str ("evt"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$debug_extension"), var_str ("debug"), &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?debug_process"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?debug_transforms"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?debug_decay"), .true., &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?debug_verbose"), .true., &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_hepevt"), var_str ("hepevt"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_ascii_short"), &
      var_str ("short.evt"), intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_ascii_long"), &
      var_str ("long.evt"), intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_athena"), &
      var_str ("athena.evt"), intrinsic=.true.)
call var_list_append_string &
      (global%var_list, var_str ("$extension_mokka"), &
       var_str ("mokka.evt"), intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$lhef_version"), var_str ("2.0"), &
     intrinsic = .true.)
call var_list_append_string &
     (global%var_list, var_str ("$lhef_extension"), var_str ("lhe"), &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?lhef_write_sqme_prc"), .true., &
     intrinsic = .true.)
call var_list_append_log &
     (global%var_list, var_str ("?lhef_write_sqme_ref"), .false., &
     intrinsic = .true.)
```

```
call var_list_append_log &
     (global%var_list, var_str ("?lhef_write_sqme_alt"), .true., &
     intrinsic = .true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_lha"), var_str ("lha"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_hepmc"), var_str ("hepmc"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_stdhep"), var_str ("stdhep"), &
      intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_stdhep_up"), &
      var_str ("up.stdhep"), intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_hepevt_verb"), &
      var_str ("hepevt.verb"), intrinsic=.true.)
call var_list_append_string &
     (global%var_list, var_str ("$extension_lha_verb"), &
      var_str ("lha.verb"), intrinsic=.true.)
call var_list_append_int (global%var_list, &
     var_str ("n_bins"), 20, &
     intrinsic=.true.)
call var_list_append_log (global%var_list, &
     var_str ("?normalize_bins"), .false., &
     intrinsic=.true.)
call var_list_append_string (global%var_list, &
     var_str ("$obs_label"), var_str (""), &
     intrinsic=.true.)
call var_list_append_string (global%var_list, &
     var_str ("$obs_unit"), var_str (""), &
      intrinsic=.true.)
call var_list_append_string (global%var_list, &
     var_str ("$title"), var_str (""), &
      intrinsic=.true.)
call var_list_append_string (global%var_list, &
     var_str ("$description"), var_str (""), &
      intrinsic=.true.)
call var_list_append_string (global%var_list, &
     var_str ("$x_label"), var_str (""), &
      intrinsic=.true.)
call var_list_append_string (global%var_list, &
     var_str ("$y_label"), var_str (""), &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("graph_width_mm"), 130, &
      intrinsic=.true.)
call var_list_append_int &
     (global%var_list, var_str ("graph_height_mm"), 90, &
      intrinsic=.true.)
call var_list_append_log &
     (global%var_list, var_str ("?y_log"), .false., &
      intrinsic=.true.)
```

2800

```
                 call var_list_append_log &
                      (global%var_list, var_str ("?x_log"), .false., &
                       intrinsic=.true.)
                 call var_list_append_real &
                      (global%var_list, var_str ("x_min"),  &
                       intrinsic=.true.)
                 call var_list_append_real &
                      (global%var_list, var_str ("x_max"),  &
                       intrinsic=.true.)
                 call var_list_append_real &
                      (global%var_list, var_str ("y_min"),  &
                       intrinsic=.true.)
                 call var_list_append_real &
                      (global%var_list, var_str ("y_max"),  &
                       intrinsic=.true.)
                 call var_list_append_string &
                      (global%var_list, var_str ("$gmlcode_bg"), var_str (""), &
                       intrinsic=.true.)
                 call var_list_append_string &
                      (global%var_list, var_str ("$gmlcode_fg"), var_str (""), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?draw_histogram"), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?draw_base"), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?draw_piecewise"), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?fill_curve"), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?draw_curve"), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?draw_errors"), &
                       intrinsic=.true.)
                 call var_list_append_log &
                      (global%var_list, var_str ("?draw_symbols"), &
                       intrinsic=.true.)
                 call var_list_append_string &
                      (global%var_list, var_str ("$fill_options"), &
                       intrinsic=.true.)
                 call var_list_append_string &
                      (global%var_list, var_str ("$draw_options"), &
                       intrinsic=.true.)
                 call var_list_append_string &
                      (global%var_list, var_str ("$err_options"), &
                       intrinsic=.true.)
                 call var_list_append_string &
                      (global%var_list, var_str ("$symbol"), &
                       intrinsic=.true.)
```

```
        call var_list_append_real (global%var_list, &
             var_str ("tolerance"), 0._default, &
              intrinsic=.true.)
        call var_list_append_int (global%var_list, &
             var_str ("checkpoint"), 0, &
             intrinsic = .true.)
        call var_list_append_log &
             (global%var_list, var_str ("?pacify"), .false., &
             intrinsic=.true.)
        call var_list_append_string &
             (global%var_list, var_str ("$out_file"), var_str (""), &
              intrinsic=.true.)
        call var_list_append_log &
             (global%var_list, var_str ("?out_advance"), .true., &
              intrinsic=.true.)
!!! JRR: WK please check (#542)
!     call var_list_append_log &
!          (global%var_list, var_str ("?out_custom"), .false., &
!           intrinsic=.true.)
!     call var_list_append_string &
!          (global%var_list, var_str ("$out_comment"), var_str ("# "), &
!           intrinsic=.true.)
!     call var_list_append_log &
!          (global%var_list, var_str ("?out_header"), .true., &
!           intrinsic=.true.)
!     call var_list_append_log &
!          (global%var_list, var_str ("?out_yerr"), .true., &
!           intrinsic=.true.)
!     call var_list_append_log &
!          (global%var_list, var_str ("?out_xerr"), .true., &
!           intrinsic=.true.)
     call var_list_append_int (global%var_list, var_str ("real_range"), &
          range (real_specimen), intrinsic = .true., locked = .true.)
     call var_list_append_int (global%var_list, var_str ("real_precision"), &
          precision (real_specimen), intrinsic = .true., locked = .true.)
     call var_list_append_real (global%var_list, var_str ("real_epsilon"), &
          epsilon (real_specimen), intrinsic = .true., locked = .true.)
     call var_list_append_real (global%var_list, var_str ("real_tiny"), &
          tiny (real_specimen), intrinsic = .true., locked = .true.)
     !!! FastJet parameters
     call var_list_append_int (global%var_list, &
          var_str ("kt_algorithm"), &
          kt_algorithm, &
          intrinsic = .true., locked = .true.)
     call var_list_append_int (global%var_list, &
          var_str ("cambridge_algorithm"), &
          cambridge_algorithm, intrinsic = .true., locked = .true.)
     call var_list_append_int (global%var_list, &
          var_str ("antikt_algorithm"), &
          antikt_algorithm, &
          intrinsic = .true., locked = .true.)
     call var_list_append_int (global%var_list, &
          var_str ("genkt_algorithm"), &
          genkt_algorithm, &
```

```
    intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("cambridge_for_passive_algorithm"), &
     cambridge_for_passive_algorithm, &
     intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("genkt_for_passive_algorithm"), &
     genkt_for_passive_algorithm, &
     intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("ee_kt_algorithm"), &
     ee_kt_algorithm, &
     intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("ee_genkt_algorithm"), &
     ee_genkt_algorithm, &
     intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("plugin_algorithm"), &
     plugin_algorithm, &
     intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("undefined_jet_algorithm"), &
     undefined_jet_algorithm, &
     intrinsic = .true., locked = .true.)
call var_list_append_int (global%var_list, &
     var_str ("jet_algorithm"), undefined_jet_algorithm, &
     intrinsic = .true.)
call var_list_append_real (global%var_list, &
     var_str ("jet_r"), 0._default, &
     intrinsic = .true.)
call var_list_append_log &
    (global%var_list, var_str ("?polarized_events"), .false., &
        intrinsic=.true.)
!!! Default settings for shower
call var_list_append_log &
    (global%var_list, var_str ("?allow_shower"), .true., &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?ps_fsr_active"), .false., &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?ps_use_PYTHIA_shower"), .false., &
        intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?ps_PYTHIA_verbose"), .false., &
        intrinsic=.true.)
call var_list_append_string &
    (global%var_list, var_str ("$ps_PYTHIA_PYGIVE"), var_str (""), &
     intrinsic=.true.)
call var_list_append_log &
    (global%var_list, var_str ("?ps_isr_active"), .false., &
        intrinsic=.true.)
call var_list_append_real (global%var_list, &
```

2803

```
      var_str ("ps_mass_cutoff"), 1._default, intrinsic = .true.)
call var_list_append_real (global%var_list, &
      var_str ("ps_fsr_lambda"), 0.29_default, intrinsic = .true.)
call var_list_append_real (global%var_list, &
      var_str ("ps_isr_lambda"), 0.29_default, intrinsic = .true.)
call var_list_append_int (global%var_list, &
      var_str ("ps_max_n_flavors"), 5, intrinsic = .true.)
call var_list_append_log &
      (global%var_list, var_str ("?ps_isr_alpha_s_running"), .true., &
         intrinsic=.true.)
call var_list_append_log &
      (global%var_list, var_str ("?ps_fsr_alpha_s_running"), .true., &
         intrinsic=.true.)
call var_list_append_real (global%var_list, var_str ("ps_fixed_alpha_s"), &
      0._default, intrinsic = .true.)
call var_list_append_log &
      (global%var_list, var_str ("?ps_isr_pt_ordered"), .false., &
         intrinsic=.true.)
call var_list_append_log &
      (global%var_list, var_str ("?ps_isr_angular_ordered"), .true., &
         intrinsic=.true.)
call var_list_append_real (global%var_list, var_str &
      ("ps_isr_primordial_kt_width"), 0._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("ps_isr_primordial_kt_cutoff"), 5._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("ps_isr_z_cutoff"), 0.999_default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("ps_isr_minenergy"), 1._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("ps_isr_tscalefactor"), 1._default, intrinsic = .true.)
call var_list_append_log (global%var_list, var_str &
      ("?ps_isr_only_onshell_emitted_partons"), .false., intrinsic=.true.)
!!! Default settings for hadronization
call var_list_append_log &
      (global%var_list, var_str ("?hadronization_active"), .false., &
         intrinsic=.true.)
!!! Setting for mlm matching
call var_list_append_log &
      (global%var_list, var_str ("?mlm_matching"), .false., &
         intrinsic=.true.)
call var_list_append_real (global%var_list, var_str &
      ("mlm_Qcut_ME"), 0._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("mlm_Qcut_PS"), 0._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("mlm_ptmin"), 0._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("mlm_etamax"), 0._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("mlm_Rmin"), 0._default, intrinsic = .true.)
call var_list_append_real (global%var_list, var_str &
      ("mlm_Emin"), 0._default, intrinsic = .true.)
call var_list_append_int (global%var_list, var_str &
```

```
                 ("mlm_nmaxMEjets"), 0, intrinsic = .true.)
        call var_list_append_real (global%var_list, var_str &
             ("mlm_ETclusfactor"), 0.2_default, intrinsic = .true.)
        call var_list_append_real (global%var_list, var_str &
             ("mlm_ETclusminE"), 5._default, intrinsic = .true.)
        call var_list_append_real (global%var_list, var_str &
             ("mlm_etaclusfactor"), 1._default, intrinsic = .true.)
        call var_list_append_real (global%var_list, var_str &
             ("mlm_Rclusfactor"), 1._default, intrinsic = .true.)
        call var_list_append_real (global%var_list, var_str &
             ("mlm_Eclusfactor"), 1._default, intrinsic = .true.)
        call var_list_append_log &
             (global%var_list, var_str ("?ckkw_matching"), .false., &
                intrinsic=.true.)
        call var_list_append_log &
             (global%var_list, var_str ("?muli_active"), .false., &
                intrinsic=.true.)
        call var_list_append_string &
             (global%var_list, var_str ("$pdf_builtin_set"), var_str ("CTEQ6L"), &
             intrinsic=.true.)
        call var_list_append_log &
             (global%var_list, var_str ("?omega_openmp"), &
             openmp_is_active (), &
             intrinsic=.true.)
        call var_list_append_log &
             (global%var_list, var_str ("?openmp_is_active"), &
             openmp_is_active (), &
             locked=.true., intrinsic=.true.)
        call var_list_append_int &
             (global%var_list, var_str ("openmp_num_threads_default"), &
             openmp_get_default_max_threads (), &
             locked=.true., intrinsic=.true.)
        call var_list_append_int &
             (global%var_list, var_str ("openmp_num_threads"), &
             openmp_get_max_threads (), &
             intrinsic=.true.)
        call var_list_append_log &
             (global%var_list, var_str ("?openmp_logging"), &
             .true., intrinsic=.true.)
        call global%init_pointer_variables ()
        call global%process_stack%init_var_list (global%var_list)
      end subroutine rt_data_global_init
```

22.3.7   Local copies

This is done at compile time when a local copy of runtime data is needed:
Link the variable list and initialize all derived parameters. This allows for
synchronizing them with local variable changes without affecting global data.

Also re-initialize pointer variables, so they point to local copies of their
targets.

⟨*RT data: rt data: TBP*⟩+≡
```
    procedure :: local_init => rt_data_local_init
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_local_init (local, global, env)
  class(rt_data_t), intent(inout), target :: local
  class(rt_data_t), intent(in), target :: global
  integer, intent(in), optional :: env
  call var_list_link (local%var_list, global%var_list)
  if (associated (global%model)) then
     call var_list_init_copies (local%var_list, &
          model_get_var_list_ptr (global%model), &
          derived_only = .true.)
  end if
  call local%init_pointer_variables ()
  local%os_data = global%os_data
  local%logfile = global%logfile
end subroutine rt_data_local_init
```

This copies global objects, for which local modifications should be kept.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: copy_globals => rt_data_copy_globals
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_copy_globals (global, local)
  class(rt_data_t), intent(in) :: global
  class(rt_data_t), intent(inout) :: local
  local%model_list = global%model_list
  local%prclib_stack = global%prclib_stack
  local%process_stack = global%process_stack
end subroutine rt_data_copy_globals
```

These variables point to objects which get local copies:

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: init_pointer_variables => rt_data_init_pointer_variables
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_init_pointer_variables (local)
  class(rt_data_t), intent(inout), target :: local
  logical, target, save :: known = .true.
  call var_list_append_string_ptr &
       (local%var_list, var_str ("$fc"), local%os_data%fc, known, &
        intrinsic=.true.)
  call var_list_append_string_ptr &
       (local%var_list, var_str ("$fcflags"), local%os_data%fcflags, known, &
        intrinsic=.true.)
end subroutine rt_data_init_pointer_variables
```

This is done at execution time: Copy data, transfer pointers. `local` has intent(inout) because its local variable list has already been prepared by the previous routine.

To be pedantic, the local pointers to model and library should point to the entries in the local copies. (However, as long as these are just shallow copies with identical content, this is actually irrelevant.)

The process library and process stacks behave as global objects. The copies of the process library and process stacks should be shallow copies, so the contents stay identical. Since objects may be pushed on the stack in the local environment, upon restoring the global environment, we should reverse the assignment. Then the added stack elements will end up on the global stack. (This should be reconsidered in a parallel environment.)

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: link => rt_data_link
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_link (local, global)
  class(rt_data_t), intent(inout), target :: local
  class(rt_data_t), intent(in), target :: global
  local%lexer => global%lexer
  call global%copy_globals (local)
  local%os_data = global%os_data
  local%logfile = global%logfile
  call var_list_link (local%var_list, global%var_list)
  if (associated (global%model)) then
     local%model => &
          local%model_list%get_model_ptr (global%model%get_name ())
     call var_list_synchronize (local%var_list, &
          model_get_var_list_ptr (local%model), reset_pointers = .true.)
     call local%particle_stack%init (local%model)
  end if
  if (associated (global%prclib)) then
     local%prclib => &
          local%prclib_stack%get_library_ptr (global%prclib%get_name ())
  end if
  local%it_list = global%it_list
  local%beam_structure = global%beam_structure
  local%lhapdf_status = global%lhapdf_status
  local%pn = global%pn
  if (allocated (local%sample_fmt))  deallocate (local%sample_fmt)
  if (allocated (global%sample_fmt)) then
     allocate (local%sample_fmt (size (global%sample_fmt)), &
          source = global%sample_fmt)
  end if
  local%out_files => global%out_files
 end subroutine rt_data_link
```

Restore the previous state of data; in particular, the variable list. This applies only to model variables (which are copies); other variables are automatically restored when local variables are removed.

If the `keep_local` flag is set, we want to retain current settings in the local environment. In particular, we create an instance of the currently selected model (which thus becomes separated from the model library!), and we synchronize the local variable list with it. The other local variables are also kept.

Some command (`read_slha`) reads in model variables as local entities. In this case, the original values of model variables should not be restored, but the global variable list should be synchronized. However, this matters only if the local model is identical to the global model; otherwise, restoring will apply to a different model.

Finally, model list, process library stack, and process stack are copied back. Thus, they behave like global objects: all changes in the local environment are kept in the global context. Note that these are shallow copies.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: restore => rt_data_restore
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_restore (global, local, keep_model_vars, keep_local)
  class(rt_data_t), intent(inout) :: global
  class(rt_data_t), intent(inout) :: local
  logical, intent(in), optional :: keep_model_vars, keep_local
  logical :: same_model, restore, delete
  delete = .true.;  if (present (keep_local))  delete = .not. keep_local
  if (delete) then
     call var_list_undefine (local%var_list, follow_link=.false.)
  else
     if (associated (local%model)) then
        call model_pointer_to_instance (local%model)
        call var_list_synchronize (local%var_list, &
             model_get_var_list_ptr (local%model), reset_pointers = .true.)
     end if
  end if
  if (associated (global%model)) then
     call local%particle_stack%restore_model ()
     same_model = &
         global%model%get_name () == local%model%get_name ()
     if (present (keep_model_vars) .and. same_model) then
        restore = .not. keep_model_vars
     else
        if (.not. same_model)  call msg_message ("Restoring model '" // &
             char (global%model%get_name ()) // "'")
        restore = .true.
     end if
     if (restore) then
        call var_list_restore (global%var_list)
     else
        call var_list_synchronize &
             (global%var_list, model_get_var_list_ptr (global%model))
     end if
  end if
  call global%restore_globals (local)
end subroutine rt_data_restore
```

This restores global objects, for which local modifications should be kept.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: restore_globals => rt_data_restore_globals
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_restore_globals (global, local)
  class(rt_data_t), intent(inout) :: global
  class(rt_data_t), intent(in) :: local
  global%model_list = local%model_list
  global%prclib_stack = local%prclib_stack
  global%process_stack = local%process_stack
```

```
        end subroutine rt_data_restore_globals
```

22.3.8   Finalization

Finalizer for the variable list and the structure-function list. This is done only
for the global RT dataset; local copies contain pointers to this and do not need
a finalizer.

⟨*RT data: rt data: TBP*⟩+≡
```
    procedure :: final => rt_data_global_final
```

⟨*RT data: procedures*⟩+≡
```
    subroutine rt_data_global_final (global)
      class(rt_data_t), intent(inout) :: global
      call global%process_stack%final ()
      call global%prclib_stack%final ()
      call global%particle_stack%final ()
      call global%model_list%final ()
      call var_list_final (global%var_list)
      if (associated (global%out_files)) then
         call file_list_final (global%out_files)
         deallocate (global%out_files)
      end if
    end subroutine rt_data_global_final
```

The local copy needs a finalizer just for the variable list, which consists of local
copies.

⟨*RT data: rt data: TBP*⟩+≡
```
    procedure :: local_final => rt_data_local_final
```

⟨*RT data: procedures*⟩+≡
```
    subroutine rt_data_local_final (local)
      class(rt_data_t), intent(inout) :: local
      call var_list_final (local%var_list)
    end subroutine rt_data_local_final
```

22.3.9   Filling Contents

Read a model, filling the model list component and assigning this as the current
model. Copy the model variables to the global var list.

⟨*RT data: rt data: TBP*⟩+≡
```
    procedure :: read_model => rt_data_read_model
```

⟨*RT data: procedures*⟩+≡
```
    subroutine rt_data_read_model (global, name, filename, synchronize)
      class(rt_data_t), intent(inout) :: global
      type(string_t), intent(in) :: name, filename
      type(var_list_t), pointer :: model_vars
      logical, intent(in), optional :: synchronize
      logical :: sync
      sync = .true.;  if (present (synchronize))  sync = synchronize
      call global%model_list%read_model &
```

```
          (name, filename, global%os_data, global%model)
      if (associated (global%model)) then
         call var_list_set_string (global%var_list, var_str ("$model_name"), &
              name, is_known = .true.)
         model_vars => model_get_var_list_ptr (global%model)
         call var_list_init_copies (global%var_list, model_vars)
         if (sync) then
            call var_list_synchronize (global%var_list, model_vars, &
                 reset_pointers = .true.)
         end if
         call global%particle_stack%reset (global%model)
      end if
   end subroutine rt_data_read_model
```

Select a model, assuming that it has already been read.

⟨*RT data: rt data: TBP*⟩+≡
```
   procedure :: select_model => rt_data_select_model
```

⟨*RT data: procedures*⟩+≡
```
   subroutine rt_data_select_model (global, name)
     class(rt_data_t), intent(inout) :: global
     type(string_t), intent(in) :: name
     type(var_list_t), pointer :: model_vars
     global%model => global%model_list%get_model_ptr (name)
     if (associated (global%model)) then
        call var_list_set_string (global%var_list, var_str ("$model_name"), &
             name, is_known = .true.)
        model_vars => model_get_var_list_ptr (global%model)
        call var_list_synchronize (global%var_list, model_vars, &
             reset_pointers = .true.)
        call global%particle_stack%reset (global%model)
     end if
   end subroutine rt_data_select_model
```

Modify particle properties. If necessary, we save the old status in the `particle_stack`
subobject, then modify the entry in the local model directly.

⟨*RT data: rt data: TBP*⟩+≡
```
   procedure :: modify_particle => rt_data_modify_particle
```

⟨*RT data: procedures*⟩+≡
```
   subroutine rt_data_modify_particle &
        (global, pdg, polarized, stable, decay, isotropic_decay, diagonal_decay)
     class(rt_data_t), intent(inout) :: global
     integer, intent(in) :: pdg
     logical, intent(in), optional :: polarized, stable
     logical, intent(in), optional :: isotropic_decay, diagonal_decay
     type(string_t), dimension(:), intent(in), optional :: decay
     if (.not. global%particle_stack%contains (pdg)) then
        call global%particle_stack%push (pdg)
     end if
     if (present (polarized)) then
        if (polarized) then
           call model_set_polarized (global%model, pdg)
        else
```

```
         call model_set_unpolarized (global%model, pdg)
      end if
   end if
   if (present (stable)) then
      if (stable) then
         call model_set_stable (global%model, pdg)
      else if (present (decay)) then
         call model_set_unstable &
              (global%model, pdg, decay, isotropic_decay, diagonal_decay)
      else
         call msg_bug ("Setting particle unstable: missing decay processes")
      end if
   end if
end subroutine rt_data_modify_particle
```

Add a library (available via a pointer of type `prclib_entry_t`) to the stack and update the pointer and variable list to the current library. The pointer association of `prclib_entry` will be discarded.

⟨*RT data: rt data: TBP*⟩+≡
```
   procedure :: add_prclib => rt_data_add_prclib
```

⟨*RT data: procedures*⟩+≡
```
   subroutine rt_data_add_prclib (global, prclib_entry)
     class(rt_data_t), intent(inout) :: global
     type(prclib_entry_t), intent(inout), pointer :: prclib_entry
     call global%prclib_stack%push (prclib_entry)
     call global%update_prclib (global%prclib_stack%get_first_ptr ())
   end subroutine rt_data_add_prclib
```

Given a pointer to a process library, make this the currently active library.

⟨*RT data: rt data: TBP*⟩+≡
```
   procedure :: update_prclib => rt_data_update_prclib
```

⟨*RT data: procedures*⟩+≡
```
   subroutine rt_data_update_prclib (global, lib)
     class(rt_data_t), intent(inout) :: global
     type(process_library_t), intent(in), target :: lib
     type(var_entry_t), pointer :: var
     global%prclib => lib
     var => var_list_get_var_ptr (global%var_list, &
         var_str ("$library_name"), follow_link = .false.)
     if (associated (var)) then
        call var_entry_set_string (var, &
             global%prclib%get_name (), is_known=.true.)
     else
        call var_list_append_string (global%var_list, &
             var_str ("$library_name"), global%prclib%get_name (), &
             intrinsic = .true.)
     end if
   end subroutine rt_data_update_prclib
```

22.3.10 Get contents

The helicity selection data are distributed among several parameters. Here, we collect them in a single record.

⟨*RT data: rt data: TBP*⟩+≡
```
    procedure :: get_helicity_selection => rt_data_get_helicity_selection
```

⟨*RT data: procedures*⟩+≡
```
    function rt_data_get_helicity_selection (rt_data) result (helicity_selection)
      class(rt_data_t), intent(in) :: rt_data
      type(helicity_selection_t) :: helicity_selection
      associate (var_list => rt_data%var_list)
        helicity_selection%active = var_list_get_lval (var_list, &
            var_str ("?helicity_selection_active"))
        if (helicity_selection%active) then
           helicity_selection%threshold = var_list_get_rval (var_list, &
                var_str ("helicity_selection_threshold"))
           helicity_selection%cutoff = var_list_get_ival (var_list, &
                var_str ("helicity_selection_cutoff"))
        end if
      end associate
    end function rt_data_get_helicity_selection
```

Show the beam setup: beam structure and relevant global variables.

⟨*RT data: rt data: TBP*⟩+≡
```
    procedure :: show_beams => rt_data_show_beams
```

⟨*RT data: procedures*⟩+≡
```
    subroutine rt_data_show_beams (rt_data, unit)
      class(rt_data_t), intent(in) :: rt_data
      integer, intent(in), optional :: unit
      type(string_t) :: s
      integer :: u
      u = output_unit (unit)
      associate (beams => rt_data%beam_structure, var_list => rt_data%var_list)
        call beams%write (u)
        if (.not. beams%asymmetric () .and. beams%get_n_beam () == 2) then
           write (u, "(2x,A,ES19.12,1x,'GeV')") "sqrts =", &
                var_list_get_rval (var_list, var_str ("sqrts"))
        end if
        if (beams%contains ("pdf_builtin")) then
           s = var_list_get_sval (var_list, var_str ("$pdf_builtin_set"))
           if (s /= "") then
              write (u, "(2x,A,1x,3A)")  "PDF set =", '"', char (s), '"'
           else
              write (u, "(2x,A,1x,A)")  "PDF set =", "[undefined]"
           end if
        end if
        if (beams%contains ("lhapdf")) then
           s = var_list_get_sval (var_list, var_str ("$lhapdf_dir"))
           if (s /= "") then
              write (u, "(2x,A,1x,3A)")  "LHAPDF dir    =", '"', char (s), '"'
           end if
           s = var_list_get_sval (var_list, var_str ("$lhapdf_file"))
           if (s /= "") then
```

```
         write (u, "(2x,A,1x,3A)")  "LHAPDF file   =", '"', char (s), '"'
         write (u, "(2x,A,1x,I0)") "LHAPDF member =", &
               var_list_get_ival (var_list, var_str ("lhapdf_member"))
      else
         write (u, "(2x,A,1x,A)")  "LHAPDF file   =", "[undefined]"
      end if
   end if
   if (beams%contains ("lhapdf_photon")) then
      s = var_list_get_sval (var_list, var_str ("$lhapdf_dir"))
      if (s /= "") then
         write (u, "(2x,A,1x,3A)")  "LHAPDF dir    =", '"', char (s), '"'
      end if
      s = var_list_get_sval (var_list, var_str ("$lhapdf_photon_file"))
      if (s /= "") then
         write (u, "(2x,A,1x,3A)")  "LHAPDF file   =", '"', char (s), '"'
         write (u, "(2x,A,1x,I0)") "LHAPDF member =", &
               var_list_get_ival (var_list, var_str ("lhapdf_member"))
         write (u, "(2x,A,1x,I0)") "LHAPDF scheme =", &
               var_list_get_ival (var_list, &
               var_str ("lhapdf_photon_scheme"))
      else
         write (u, "(2x,A,1x,A)")  "LHAPDF file    =", "[undefined]"
      end if
   end if
   if (beams%contains ("isr")) then
      write (u, "(2x,A,ES19.12)") "ISR alpha =", &
           var_list_get_rval (var_list, var_str ("isr_alpha"))
      write (u, "(2x,A,ES19.12)") "ISR Q max =", &
           var_list_get_rval (var_list, var_str ("isr_q_max"))
      write (u, "(2x,A,ES19.12)") "ISR mass  =", &
           var_list_get_rval (var_list, var_str ("isr_mass"))
      write (u, "(2x,A,1x,I0)") "ISR order  =", &
           var_list_get_ival (var_list, var_str ("isr_order"))
      write (u, "(2x,A,1x,L1)") "ISR recoil =", &
           var_list_get_lval (var_list, var_str ("?isr_recoil"))
   end if
   if (beams%contains ("epa")) then
      write (u, "(2x,A,ES19.12)") "EPA alpha  =", &
           var_list_get_rval (var_list, var_str ("epa_alpha"))
      write (u, "(2x,A,ES19.12)") "EPA x min  =", &
           var_list_get_rval (var_list, var_str ("epa_x_min"))
      write (u, "(2x,A,ES19.12)") "EPA Q min  =", &
           var_list_get_rval (var_list, var_str ("epa_q_min"))
      write (u, "(2x,A,ES19.12)") "EPA E max  =", &
           var_list_get_rval (var_list, var_str ("epa_e_max"))
      write (u, "(2x,A,ES19.12)") "EPA mass   =", &
           var_list_get_rval (var_list, var_str ("epa_mass"))
      write (u, "(2x,A,1x,L1)") "EPA recoil =", &
           var_list_get_lval (var_list, var_str ("?epa_recoil"))
   end if
   if (beams%contains ("ewa")) then
      write (u, "(2x,A,ES19.12)") "EWA x min      =", &
           var_list_get_rval (var_list, var_str ("ewa_x_min"))
      write (u, "(2x,A,ES19.12)") "EWA Pt max     =", &
```

```
            var_list_get_rval (var_list, var_str ("ewa_pt_max"))
       write (u, "(2x,A,ES19.12)") "EWA mass         =", &
            var_list_get_rval (var_list, var_str ("ewa_mass"))
       write (u, "(2x,A,1x,L1)") "EWA mom cons.    =", &
            var_list_get_lval (var_list, &
            var_str ("?ewa_keep_momentum"))
       write (u, "(2x,A,1x,L1)") "EWA energ. cons. =", &
            var_list_get_lval (var_list, &
            var_str ("ewa_keep_energy"))
    end if
    if (beams%contains ("circe1")) then
       write (u, "(2x,A,1x,I0)") "CIRCE1 version    =", &
            var_list_get_ival (var_list, var_str ("circe1_ver"))
       write (u, "(2x,A,1x,I0)") "CIRCE1 revision   =", &
            var_list_get_ival (var_list, var_str ("circe1_rev"))
       s = var_list_get_sval (var_list, var_str ("$circe1_acc"))
       write (u, "(2x,A,1x,A)") "CIRCE1 acceler.   =", char (s)
       write (u, "(2x,A,1x,I0)") "CIRCE1 chattin.   =", &
            var_list_get_ival (var_list, var_str ("circe1_chat"))
       write (u, "(2x,A,ES19.12)") "CIRCE1 sqrts      =", &
            var_list_get_rval (var_list, var_str ("circe1_sqrts"))
       write (u, "(2x,A,ES19.12)") "CIRCE1 epsil.     =", &
            var_list_get_rval (var_list, var_str ("circe1_eps"))
       write (u, "(2x,A,1x,L1)") "CIRCE1 phot. 1  =", &
            var_list_get_lval (var_list, var_str ("?circe1_photon1"))
       write (u, "(2x,A,1x,L1)") "CIRCE1 phot. 2  =", &
            var_list_get_lval (var_list, var_str ("?circe1_photon2"))
       write (u, "(2x,A,1x,L1)") "CIRCE1 generat. =", &
            var_list_get_lval (var_list, var_str ("?circe1_generate"))
       write (u, "(2x,A,1x,L1)") "CIRCE1 mapping  =", &
            var_list_get_lval (var_list, var_str ("?circe1_map"))
       write (u, "(2x,A,ES19.12)") "CIRCE1 map. slope =", &
            var_list_get_rval (var_list, var_str ("circe1_mapping_slope"))
    end if
    if (beams%contains ("circe2")) then
       s = var_list_get_sval (var_list, var_str ("$circe2_design"))
       write (u, "(2x,A,1x,A)") "CIRCE2 design   =", char (s)
       s = var_list_get_sval (var_list, var_str ("$circe2_file"))
       write (u, "(2x,A,1x,A)") "CIRCE2 file     =", char (s)
       write (u, "(2x,A,1x,L1)") "CIRCE2 polarized =", &
            var_list_get_lval (var_list, var_str ("?circe2_polarized"))
    end if
    if (beams%contains ("beam_events")) then
       s = var_list_get_sval (var_list, var_str ("$beam_events_file"))
       write (u, "(2x,A,1x,A)") "Beam events file     =", char (s)
       write (u, "(2x,A,1x,L1)") "Beam events EOF warn =", &
            var_list_get_lval (var_list, var_str ("?beam_events_warn_eof"))
    end if
  end associate
end subroutine rt_data_show_beams
```

Return the collision energy as determined by the current beam settings. Without beam setup, this is the `sqrts` variable.

If the value is meaningless for a setup, the function returns zero.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: get_sqrts => rt_data_get_sqrts
```

⟨*RT data: procedures*⟩+≡
```
function rt_data_get_sqrts (rt_data) result (sqrts)
  class(rt_data_t), intent(in) :: rt_data
  real(default) :: sqrts
  sqrts = var_list_get_rval (rt_data%var_list, var_str ("sqrts"))
end function rt_data_get_sqrts
```

For testing purposes, the `rt_data_t` contents can be pacified to suppress numerical fluctuations in (constant) test matrix elements.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: pacify => rt_data_pacify
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_pacify (rt_data, efficiency_reset, error_reset)
  class(rt_data_t), intent(inout) :: rt_data
  logical, intent(in), optional :: efficiency_reset, error_reset
  type(process_entry_t), pointer :: process
  process => rt_data%process_stack%first
  do while (associated (process))
     call process%pacify (efficiency_reset, error_reset)
     process => process%next
  end do
end subroutine rt_data_pacify
```

22.3.11   Test

This is the master for calling self-test procedures.

⟨*RT data: public*⟩+≡
```
public :: rt_data_test
```

⟨*RT data: tests*⟩≡
```
subroutine rt_data_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*RT data: execute tests*⟩
```
end subroutine rt_data_test
```

Initial content

Force system-dependent objects to well-defined values. Some of the variables are locked and therefore must be addressed directly.

This is, of course, only required for testing purposes. In principle, the `real_specimen` variables could be set to their values in `rt_data_t`, but this depends on the precision again, so we set them to some dummy values.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: fix_system_dependencies => rt_data_fix_system_dependencies
```

⟨*RT data: tests*⟩+≡
```
    subroutine rt_data_fix_system_dependencies (rt_data)
      class(rt_data_t), intent(inout), target :: rt_data
      type(var_list_t), pointer :: var_list
      type(var_entry_t), pointer :: var
      var_list => rt_data%var_list

      call var_list_set_log (var_list, &
           var_str ("?omega_openmp"), .false., is_known = .true.)

      var => var_list_get_var_ptr (var_list, &
           var_str ("?openmp_is_active"), V_LOG)
      call var_entry_set_log (var, .false., is_known = .true.)
      var => var_list_get_var_ptr (var_list, &
           var_str ("openmp_num_threads_default"), V_INT)
      call var_entry_set_int (var, 1, is_known = .true.)
      call var_list_set_int (var_list, &
           var_str ("openmp_num_threads"), 1, is_known = .true.)
      var => var_list_get_var_ptr (var_list, &
           var_str ("real_range"), V_INT)
      call var_entry_set_int (var, 307, is_known = .true.)
      var => var_list_get_var_ptr (var_list, &
           var_str ("real_precision"), V_INT)
      call var_entry_set_int (var, 15, is_known = .true.)
      var => var_list_get_var_ptr (var_list, &
           var_str ("real_epsilon"), V_REAL)
      call var_entry_set_real (var, 1.e-16_default, is_known = .true.)
      var => var_list_get_var_ptr (var_list, &
           var_str ("real_tiny"), V_REAL)
      call var_entry_set_real (var, 1.e-300_default, is_known = .true.)

      rt_data%os_data%fc = "Fortran-compiler"
      rt_data%os_data%fcflags = "Fortran-flags"

    end subroutine rt_data_fix_system_dependencies
```

Display the RT data in the state just after (global) initialization.

⟨*RT data: execute tests*⟩≡
```
    call test (rt_data_1, "rt_data_1", &
         "initialize", &
         u, results)
```
⟨*RT data: tests*⟩+≡
```
    subroutine rt_data_1 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: rt_data

      write (u, "(A)")  "* Test output: rt_data_1"
      write (u, "(A)")  "*   Purpose: initialize global runtime data"
      write (u, "(A)")

      call rt_data%global_init (logfile = var_str ("rt_data.log"))

      call rt_data%fix_system_dependencies ()
```

2816

```
        call var_list_set_int (rt_data%var_list, var_str ("seed"), &
             0, is_known=.true.)

        call rt_data%it_list%init ([2, 3], [5000, 20000])

        call rt_data%write (u)

        call rt_data%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: rt_data_1"

     end subroutine rt_data_1
```

Fill values

Fill in empty slots in the runtime data block.

⟨*RT data: execute tests*⟩+≡
```
     call test (rt_data_2, "rt_data_2", &
          "fill", &
          u, results)
```
⟨*RT data: tests*⟩+≡
```
     subroutine rt_data_2 (u)
       integer, intent(in) :: u
       type(rt_data_t), target :: rt_data
       type(flavor_t), dimension(2) :: flv
       type(string_t) :: cut_expr_text
       type(ifile_t) :: ifile
       type(stream_t) :: stream
       type(parse_tree_t) :: parse_tree

       write (u, "(A)")  "* Test output: rt_data_2"
       write (u, "(A)")  "*   Purpose: initialize global runtime data &
            &and fill contents"
       write (u, "(A)")

       call syntax_model_file_init ()

       call rt_data%global_init ()
       call rt_data%fix_system_dependencies ()

       call rt_data%read_model (var_str ("Test"), var_str ("Test.mdl"))

       call var_list_set_real (rt_data%var_list, var_str ("sqrts"),&
            1000._default, is_known = .true.)
       call var_list_set_int (rt_data%var_list, var_str ("seed"), &
            0, is_known=.true.)
       call flavor_init (flv, [25,25], rt_data%model)

       call var_list_set_string (rt_data%var_list, var_str ("$run_id"), &
            var_str ("run1"), is_known = .true.)
       call var_list_set_real (rt_data%var_list, var_str ("luminosity"), &
```

```
          33._default, is_known = .true.)

      call syntax_pexpr_init ()
      cut_expr_text = "all Pt > 100 [s]"
      call ifile_append (ifile, cut_expr_text)
      call stream_init (stream, ifile)
      call parse_tree_init_lexpr (parse_tree, stream, .true.)
      rt_data%pn%cuts_lexpr => parse_tree_get_root_ptr (parse_tree)

      allocate (rt_data%sample_fmt (2))
      rt_data%sample_fmt(1) = "foo_fmt"
      rt_data%sample_fmt(2) = "bar_fmt"

      call rt_data%write (u)

      call parse_tree_final (parse_tree)
      call stream_final (stream)
      call ifile_final (ifile)
      call syntax_pexpr_final ()

      call rt_data%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: rt_data_2"

    end subroutine rt_data_2
```

Save and restore

Set up a local runtime data block, change some contents, restore the global block.

For a model variable, the save-restore procedure is nontrivial. We first allocate a local copy of this variable and set a pointer to the original model variable. We set a new value, update the dependent variables and synchronize them with the local variable list, if modified. When restoring the old data, we recover the model variables from the stored values and delete the local instances.

⟨*RT data: execute tests*⟩+≡
```
    call test (rt_data_3, "rt_data_3", &
         "save/restore", &
         u, results)
```

⟨*RT data: tests*⟩+≡
```
    subroutine rt_data_3 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: rt_data, local
      type(var_list_t), pointer :: model_vars
      type(var_entry_t), pointer :: var
      type(flavor_t), dimension(2) :: flv
      type(string_t) :: cut_expr_text
      type(ifile_t) :: ifile
      type(stream_t) :: stream
```

2818

```
type(parse_tree_t) :: parse_tree
type(prclib_entry_t), pointer :: lib

write (u, "(A)")  "* Test output: rt_data_3"
write (u, "(A)")  "*   Purpose: initialize global runtime data &
     &and fill contents;"
write (u, "(A)")  "*           copy to local block and back"
write (u, "(A)")

write (u, "(A)")  "* Init global data"
write (u, "(A)")

call syntax_model_file_init ()

call rt_data%global_init ()
call rt_data%fix_system_dependencies ()
call var_list_set_int (rt_data%var_list, var_str ("seed"), &
     0, is_known=.true.)

call rt_data%read_model (var_str ("Test"), var_str ("Test.mdl"))

call var_list_set_real (rt_data%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call flavor_init (flv, [25,25], rt_data%model)

call rt_data%beam_structure%init_sf (flavor_get_name (flv), [1])
call rt_data%beam_structure%set_sf (1, 1, var_str ("pdf_builtin"))

call var_list_set_string (rt_data%var_list, var_str ("$run_id"), &
     var_str ("run1"), is_known = .true.)
call var_list_set_real (rt_data%var_list, var_str ("luminosity"), &
     33._default, is_known = .true.)

call syntax_pexpr_init ()
cut_expr_text = "all Pt > 100 [s]"
call ifile_append (ifile, cut_expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (parse_tree, stream, .true.)
rt_data%pn%cuts_lexpr => parse_tree_get_root_ptr (parse_tree)

allocate (rt_data%sample_fmt (2))
rt_data%sample_fmt(1) = "foo_fmt"
rt_data%sample_fmt(2) = "bar_fmt"

allocate (lib)
call lib%init (var_str ("library_1"))
call rt_data%add_prclib (lib)

write (u, "(A)")  "* Init and modify local data"
write (u, "(A)")

call local%local_init (rt_data)
call local%link (rt_data)
```

```
write (u, "(1x,A,L1)")  "model associated   = ", associated (local%model)
write (u, "(1x,A,L1)")  "library associated = ", associated (local%prclib)
write (u, *)

var => var_list_get_var_ptr (local%var_list, var_str ("ms"))
if (var_entry_is_copy (var)) then
   call var_list_init_copy (local%var_list, var, user=.true.)
   model_vars => model_get_var_list_ptr (local%model)
   call var_list_set_original_pointer (local%var_list, var_str ("ms"), &
       model_vars)
   call var_list_set_real (local%var_list, var_str ("ms"), &
     150._default, is_known = .true., model_name = var_str ("Test"))
   call model_parameters_update (local%model)
   call var_list_synchronize (local%var_list, model_vars)
end if

call var_list_append_string (local%var_list, &
    var_str ("$integration_method"), intrinsic = .true., user = .true.)
call var_list_set_string (local%var_list, var_str ("$integration_method"), &
    var_str ("midpoint"), is_known = .true.)

call var_list_append_string (local%var_list, &
    var_str ("$phs_method"), intrinsic = .true., user = .true.)
call var_list_set_string (local%var_list, var_str ("$phs_method"), &
    var_str ("single"), is_known = .true.)

local%os_data%fc = "Local compiler"

allocate (lib)
call lib%init (var_str ("library_2"))
call local%add_prclib (lib)

call local%write (u)

write (u, "(A)")
write (u, "(A)")  "* Restore global data"
write (u, "(A)")

call rt_data%restore (local)

write (u, "(1x,A,L1)")  "model associated   = ", associated (rt_data%model)
write (u, "(1x,A,L1)")  "library associated = ", associated (rt_data%prclib)
write (u, *)

call rt_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call parse_tree_final (parse_tree)
call stream_final (stream)
call ifile_final (ifile)
call syntax_pexpr_final ()
```

```
    call rt_data%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_3"

  end subroutine rt_data_3
```

Show variables

Display selected variables in the global record.

⟨*RT data: execute tests*⟩+≡
```
    call test (rt_data_4, "rt_data_4", &
         "show variables", &
         u, results)
```

⟨*RT data: tests*⟩+≡
```
  subroutine rt_data_4 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: rt_data

    type(string_t), dimension(0) :: empty_string_array

    write (u, "(A)")  "* Test output: rt_data_4"
    write (u, "(A)")  "*   Purpose: display selected variables"
    write (u, "(A)")

    call rt_data%global_init ()

    write (u, "(A)")  "* No variables:"
    write (u, "(A)")

    call rt_data%write_vars (u, empty_string_array)

    write (u, "(A)")  "* Two variables:"
    write (u, "(A)")

    call rt_data%write_vars (u, &
         [var_str ("?unweighted"), var_str ("$phs_method")])

    write (u, "(A)")
    write (u, "(A)")  "* Display whole record with selected variables"
    write (u, "(A)")

    call rt_data%write (u, &
         vars = [var_str ("?unweighted"), var_str ("$phs_method")])

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_4"

  end subroutine rt_data_4
```

Show parts

Display only selected parts in the state just after (global) initialization.

⟨RT data: execute tests⟩+≡
```
call test (rt_data_5, "rt_data_5", &
     "show parts", &
     u, results)
```

⟨RT data: tests⟩+≡
```
subroutine rt_data_5 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: rt_data

  write (u, "(A)")  "* Test output: rt_data_5"
  write (u, "(A)")  "*   Purpose: display parts of rt data"
  write (u, "(A)")

  call rt_data%global_init ()
  call rt_data%write_libraries (u)

  write (u, "(A)")

  call rt_data%write_beams (u)

  write (u, "(A)")

  call rt_data%write_process_stack (u)

  call rt_data%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: rt_data_5"

end subroutine rt_data_5
```

Local Model

Locally modify a model and restore the global one. We need an auxiliary function to determine the status of a model particle:

⟨RT data: tests⟩+≡
```
function is_stable (pdg, rt_data) result (flag)
  integer, intent(in) :: pdg
  type(rt_data_t), intent(in) :: rt_data
  logical :: flag
  type(flavor_t) :: flv
  call flavor_init (flv, pdg, rt_data%model)
  flag = flavor_is_stable (flv)
end function is_stable

function is_polarized (pdg, rt_data) result (flag)
  integer, intent(in) :: pdg
  type(rt_data_t), intent(in) :: rt_data
  logical :: flag
```

```
      type(flavor_t) :: flv
      call flavor_init (flv, pdg, rt_data%model)
      flag = flavor_is_polarized (flv)
    end function is_polarized
```

⟨*RT data: execute tests*⟩+≡
```
    call test (rt_data_6, "rt_data_6", &
         "local model", &
         u, results)
```
⟨*RT data: tests*⟩+≡
```
    subroutine rt_data_6 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: rt_data, local
      type(var_list_t), pointer :: model_vars
      type(var_entry_t), pointer :: var_entry
      type(string_t) :: var_name

      write (u, "(A)")  "* Test output: rt_data_6"
      write (u, "(A)")  "*   Purpose: apply and keep local modifications to model"
      write (u, "(A)")

      call syntax_model_file_init ()

      call rt_data%global_init ()
      call rt_data%read_model (var_str ("Test"), var_str ("Test.mdl"), &
           synchronize=.true.)

      write (u, "(A)")  "* Original model"
      write (u, "(A)")

      call rt_data%write_model_list (u)
      write (u, *)
      write (u, "(A,L1)")  "s is stable    = ", is_stable (25, rt_data)
      write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, rt_data)

      write (u, *)

      var_name = "ff"

      write (u, "(A)", advance="no")  "Global model variable: "
      model_vars => model_get_var_list_ptr (rt_data%model)
      call var_list_write_var (model_vars, var_name, u)

      write (u, "(A)", advance="no")  "Global variable: "
      call var_list_write_var (rt_data%var_list, var_name, u)

      write (u, "(A)")
      write (u, "(A)")  "* Apply local modifications: unstable"
      write (u, "(A)")

      call local%local_init (rt_data)
      call local%link (rt_data)
```

```
var_entry => var_list_get_var_ptr &
   (rt_data%var_list, var_name, V_REAL, follow_link=.false.)
call var_list_init_copy (local%var_list, var_entry, user=.true.)

call var_list_set_original_pointer (local%var_list, var_name, &
     model_get_var_list_ptr (local%model))
call var_list_set_real (local%var_list, var_name, 0.4_default, &
     is_known = .true., verbose = .true., model_name = var_str ("Test"))
call var_list_restore (local%var_list)
call model_parameters_update (local%model)

call local%modify_particle (25, stable = .false., decay = [var_str ("d1")])
call local%modify_particle (6, stable = .false., &
     decay = [var_str ("f1")], isotropic_decay = .true.)
call local%modify_particle (-6, stable = .false., &
     decay = [var_str ("f2"), var_str ("f3")], diagonal_decay = .true.)

call model_write (local%model, u)

write (u, "(A)")
write (u, "(A)")  "* Further modifications"
write (u, "(A)")

call local%modify_particle (6, stable = .false., &
     decay = [var_str ("f1")], &
     diagonal_decay = .true., isotropic_decay = .false.)
call local%modify_particle (-6, stable = .false., &
     decay = [var_str ("f2"), var_str ("f3")], &
     diagonal_decay = .false., isotropic_decay = .true.)
call model_write (local%model, u)

write (u, "(A)")
write (u, "(A)")  "* Further modifications: f stable but polarized"
write (u, "(A)")

call local%modify_particle (6, stable = .true., polarized = .true.)
call local%modify_particle (-6, stable = .true.)
call model_write (local%model, u)

write (u, *)

call local%particle_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Global model"
write (u, "(A)")

call model_write (local%model, u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, rt_data)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, rt_data)

write (u, "(A)")
write (u, "(A)")  "* Local model"
```

```
write (u, "(A)")

call model_write (local%model, u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, local)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, local)

write (u, *)

write (u, "(A)", advance="no")  "Global model variable: "
model_vars => model_get_var_list_ptr (rt_data%model)
call var_list_write_var (model_vars, var_name, u)

write (u, "(A)", advance="no")  "Local model variable: "
call var_list_write_var (model_get_var_list_ptr (local%model), &
     var_name, u)

write (u, "(A)", advance="no")  "Global variable: "
call var_list_write_var (rt_data%var_list, var_name, u)

write (u, "(A)", advance="no")  "Local variable: "
call var_list_write_var (local%var_list, var_name, u)

write (u, "(A)")
write (u, "(A)")  "* Restore global"

call rt_data%restore (local, keep_local = .true.)

write (u, "(A)")
write (u, "(A)")  "* Global model"
write (u, "(A)")

call model_write (rt_data%model, u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, rt_data)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, rt_data)

write (u, "(A)")
write (u, "(A)")  "* Local model"
write (u, "(A)")

call model_write (local%model, u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, local)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, local)

write (u, *)

write (u, "(A)", advance="no")  "Global model variable: "
model_vars => model_get_var_list_ptr (rt_data%model)
call var_list_write_var (model_vars, var_name, u)

write (u, "(A)", advance="no")  "Local model variable: "
call var_list_write_var (model_get_var_list_ptr (local%model), &
```

```
          var_name, u)

     write (u, "(A)", advance="no")  "Global variable: "
     call var_list_write_var (rt_data%var_list, var_name, u)

     write (u, "(A)", advance="no")  "Local variable: "
     call var_list_write_var (local%var_list, var_name, u)

     write (u, "(A)")
     write (u, "(A)")  "* Cleanup"

     call model_final (local%model)
     deallocate (local%model)

     call rt_data%final ()
     call syntax_model_file_final ()

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: rt_data_6"

   end subroutine rt_data_6
```

Result variables

Initialize result variables and check that they are accessible via the global variable list.

⟨*RT data: execute tests*⟩+≡
```
     call test (rt_data_7, "rt_data_7", &
          "result variables", &
          u, results)
```

⟨*RT data: tests*⟩+≡
```
   subroutine rt_data_7 (u)
     integer, intent(in) :: u
     type(rt_data_t), target :: rt_data

     write (u, "(A)")  "* Test output: rt_data_7"
     write (u, "(A)")  "*   Purpose: set and access result variables"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize process variables"
     write (u, "(A)")

     call rt_data%global_init ()
     call rt_data%process_stack%init_result_vars (var_str ("testproc"))

     call var_list_write_var (rt_data%var_list, &
          var_str ("integral(testproc)"), u)
     call var_list_write_var (rt_data%var_list, &
          var_str ("error(testproc)"), u)

     write (u, "(A)")
     write (u, "(A)")  "* Cleanup"
```

```
     call rt_data%final ()

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: rt_data_7"

  end subroutine rt_data_7
```

Beam energy

If beam parameters are set, the variable `sqrts` is not necessarily the collision energy. The method `get_sqrts` fetches the correct value.

⟨*RT data: execute tests*⟩+≡
```
     call test (rt_data_8, "rt_data_8", &
          "beam energy", &
          u, results)
```

⟨*RT data: tests*⟩+≡
```
  subroutine rt_data_8 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: rt_data

    write (u, "(A)")  "* Test output: rt_data_8"
    write (u, "(A)")  "*   Purpose: get correct collision energy"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    call rt_data%global_init ()

    write (u, "(A)")  "* Set sqrts"
    write (u, "(A)")

    call var_list_set_real (rt_data%var_list, var_str ("sqrts"), &
         1000._default, is_known = .true.)
    write (u, "(1x,A,ES19.12)")  "sqrts =", rt_data%get_sqrts ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rt_data%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_8"

  end subroutine rt_data_8
```

22.4 Select implementations

For abstract types (process core, integrator, phase space, etc.), we need a way to dynamically select a concrete type, using either data given by the user or a previous selection of a concrete type. This is done by subroutines in the current module.

This module provides no new types, just procedures.

⟨dispatch.f90⟩≡
  ⟨*File header*⟩

  module dispatch

  ⟨*Use kinds*⟩
    use kinds, only: i16 !NODEP!
  ⟨*Use strings*⟩
    use limits, only: MZ_REF, ALPHA_QCD_MZ_REF !NODEP!
    use constants, only: PI !NODEP!
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use os_interface
    use unit_tests
    use sm_qcd
    use pdg_arrays
    use variables
    use flavors
    use interactions
    use models
    use process_constants
    use pdf_builtin !NODEP!
    use sf_mappings
    use sf_base
    use sf_pdf_builtin
    use sf_lhapdf
    use sf_circe1
    use sf_circe2
    use sf_isr
    use sf_epa
    use sf_ewa
    use sf_escan
    use sf_beam_events
    use sf_user
    use rng_base
    use rng_tao
    use mci_base
    use mci_midpoint
    use mci_vamp
    use phs_base
    use mappings
    use phs_forests
    use phs_single
    use phs_wood
    use prc_core_def
    use prc_test

```
    use beams
    use prc_omega
    use prc_template_me
    use prc_core
    use processes
    use shower
    use event_transforms
    use decays
    use beam_structures
    use eio_base
    use eio_raw
    use eio_checkpoints
    use eio_lhef
    use eio_hepmc
    use eio_stdhep
    use eio_ascii
    use eio_weights
    use rt_data
```

⟨*Standard module head*⟩

⟨*Dispatch: public*⟩

⟨*Dispatch: types*⟩

```
contains
```

⟨*Dispatch: procedures*⟩

⟨*Dispatch: tests*⟩

```
end module dispatch
```

22.4.1   Process Core Definition

The `prc_core_def_t` abstract type can be instantiated by providing a `$method`
string variable.

Note: `core_def` has intent(inout) because gfortran 4.7.1 crashes for intent(out).

⟨*Dispatch: public*⟩≡
```
    public :: dispatch_core_def
```

⟨*Dispatch: procedures*⟩≡
```
    subroutine dispatch_core_def (core_def, prt_in, prt_out, global)
      class(prc_core_def_t), allocatable, intent(inout) :: core_def
      type(string_t), dimension(:), intent(in) :: prt_in
      type(string_t), dimension(:), intent(in) :: prt_out
      type(rt_data_t), intent(in) :: global
      type(string_t) :: method
      type(string_t) :: model_name
      type(string_t) :: restrictions
      logical :: openmp_support
      logical :: report_progress
      logical :: diags, diags_color
```

2829

```fortran
    type(string_t) :: extra_options
    type(model_t), pointer :: model
    model => global%model
    associate (var_list => global%var_list)
      method = var_list_get_sval (var_list, var_str ("$method"))
      model_name = var_list_get_sval (var_list, var_str ("$model_name"))
      select case (char (method))
      case ("unit_test")
         allocate (prc_test_def_t :: core_def)
         select type (core_def)
         type is (prc_test_def_t)
            call core_def%init (model_name, prt_in, prt_out)
         end select
      case ("template")
         allocate (template_me_def_t :: core_def)
         select type (core_def)
         type is (template_me_def_t)
            call core_def%init (model, prt_in, prt_out, unity = .false.)
         end select
      case ("template_unity")
         allocate (template_me_def_t :: core_def)
         select type (core_def)
         type is (template_me_def_t)
            call core_def%init (model, prt_in, prt_out, unity = .true.)
         end select
      case ("omega")
         diags = var_list_get_lval (var_list, &
              var_str ("?diags"))
         diags_color = var_list_get_lval (var_list, &
              var_str ("?diags_color"))
         restrictions = var_list_get_sval (var_list, &
              var_str ("$restrictions"))
         openmp_support = var_list_get_lval (var_list, &
              var_str ("?omega_openmp"))
         report_progress = var_list_get_lval (var_list, &
              var_str ("?report_progress"))
         extra_options = var_list_get_sval (var_list, &
              var_str ("$omega_flags"))
         allocate (omega_def_t :: core_def)
         select type (core_def)
         type is (omega_def_t)
            call core_def%init (model_name, prt_in, prt_out, &
                 restrictions, openmp_support, report_progress, &
                 extra_options, diags, diags_color)
         end select
      case default
         call msg_fatal ("Process configuration: method '" &
              // char (method) // "' not implemented")
      end select
    end associate
  end subroutine dispatch_core_def
```

22.4.2 Process core allocation

Here we allocate an object of abstract type `prc_core_t` with a concrete type that matches a process definition. The `prc_omega_t` extension will require the current parameter set, so we take the opportunity to grab it from the model.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_core
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_core (core, core_def, model, &
        helicity_selection, qcd, use_color_factors)
      class(prc_core_t), allocatable, intent(inout) :: core
      class(prc_core_def_t), intent(in) :: core_def
      type(model_t), intent(in), target, optional :: model
      type(helicity_selection_t), intent(in), optional :: helicity_selection
      type(qcd_t), intent(in), optional :: qcd
      logical, intent(in), optional :: use_color_factors
      select type (core_def)
      type is (prc_test_def_t)
         allocate (test_t :: core)
      type is (template_me_def_t)
         allocate (prc_template_me_t :: core)
         select type (core)
         type is (prc_template_me_t)
            call core%set_parameters (model)
         end select
      type is (omega_def_t)
         allocate (prc_omega_t :: core)
         select type (core)
         type is (prc_omega_t)
            call core%set_parameters (model, &
                 helicity_selection, qcd, use_color_factors)
         end select
      class default
         call msg_bug ("Process core: unexpected process definition type")
      end select
    end subroutine dispatch_core
```

22.4.3 Process core update and restoration

Here we take an existing object of abstract type `prc_core_t` and update the parameters as given by the current state of `model`. Optionally, we can save the previous state as `saved_core`. The second routine restores the original from the save.

(In the test case, there is no possible update.)

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_core_update
    public :: dispatch_core_restore
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_core_update (core, model, helicity_selection, qcd, &
        saved_core)
      class(prc_core_t), allocatable, intent(inout) :: core
```

```
      type(model_t), intent(in), optional, target :: model
      type(helicity_selection_t), intent(in), optional :: helicity_selection
      type(qcd_t), intent(in), optional :: qcd
      class(prc_core_t), allocatable, intent(inout), optional :: saved_core
      if (present (saved_core)) then
         allocate (saved_core, source = core)
      end if
      select type (core)
      type is (test_t)
      type is (prc_omega_t)
         call core%set_parameters (model, helicity_selection, qcd)
         call core%activate_parameters ()
      class default
         call msg_bug ("Process core update: unexpected process definition type")
      end select
    end subroutine dispatch_core_update

    subroutine dispatch_core_restore (core, saved_core)
      class(prc_core_t), allocatable, intent(inout) :: core
      class(prc_core_t), allocatable, intent(inout) :: saved_core
      call move_alloc (from = saved_core, to = core)
      select type (core)
      type is (test_t)
      type is (prc_omega_t)
         call core%activate_parameters ()
      class default
         call msg_bug ("Process core restore: unexpected process definition type")
      end select
    end subroutine dispatch_core_restore
```

22.4.4 Integrator allocation

Allocate an integrator according to the variable `$integration_method`.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_mci
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_mci (mci, global, process_id)
      class(mci_t), allocatable, intent(inout) :: mci
      type(rt_data_t), intent(in) :: global
      type(string_t), intent(in) :: process_id
      type(string_t) :: run_id
      type(string_t) :: integration_method
      type(grid_parameters_t) :: grid_par
      type(history_parameters_t) :: history_par
      logical :: rebuild_grids, check_grid_file, negative_weights, verbose
      integration_method = var_list_get_sval (global%var_list, &
           var_str ("$integration_method"))
      select case (char (integration_method))
      case ("midpoint")
         allocate (mci_midpoint_t :: mci)
      case ("vamp", "default")
         associate (var_list => global%var_list)
```

```
      grid_par%threshold_calls = &
           var_list_get_ival (var_list, var_str ("threshold_calls"))
      grid_par%min_calls_per_channel = &
           var_list_get_ival (var_list, var_str ("min_calls_per_channel"))
      grid_par%min_calls_per_bin = &
           var_list_get_ival (var_list, var_str ("min_calls_per_bin"))
      grid_par%min_bins = &
           var_list_get_ival (var_list, var_str ("min_bins"))
      grid_par%max_bins = &
           var_list_get_ival (var_list, var_str ("max_bins"))
      grid_par%stratified = &
           var_list_get_lval (var_list, var_str ("?stratified"))
      grid_par%use_vamp_equivalences = &
           var_list_get_lval (var_list, var_str ("?use_vamp_equivalences"))
      grid_par%channel_weights_power = &
           var_list_get_rval (var_list, var_str ("channel_weights_power"))
      grid_par%accuracy_goal = &
           var_list_get_rval (var_list, var_str ("accuracy_goal"))
      grid_par%error_goal = &
           var_list_get_rval (var_list, var_str ("error_goal"))
      grid_par%rel_error_goal = &
           var_list_get_rval (var_list, var_str ("relative_error_goal"))
      history_par%global = &
           var_list_get_lval (var_list, &
           var_str ("?vamp_history_global"))
      history_par%global_verbose = &
           var_list_get_lval (var_list, &
           var_str ("?vamp_history_global_verbose"))
      history_par%channel = &
           var_list_get_lval (var_list, &
           var_str ("?vamp_history_channels"))
      history_par%channel_verbose = &
           var_list_get_lval (var_list, &
           var_str ("?vamp_history_channels_verbose"))
      verbose = var_list_get_lval (var_list, var_str ("?vamp_verbose"))
      check_grid_file = &
           var_list_get_lval (var_list, var_str ("?check_grid_file"))
      run_id = &
           var_list_get_sval (var_list, var_str ("$run_id"))
      rebuild_grids = &
           var_list_get_lval (var_list, var_str ("?rebuild_grids"))
      negative_weights = &
           var_list_get_lval (var_list, var_str ("?negative_weights"))
    end associate
    allocate (mci_vamp_t :: mci)
    select type (mci)
    type is (mci_vamp_t)
       call mci%set_grid_parameters (grid_par)
       if (run_id /= "") then
          call mci%set_grid_filename (process_id, run_id)
       else
          call mci%set_grid_filename (process_id)
       end if
       call mci%set_history_parameters (history_par)
```

2833

```
      call mci%set_rebuild_flag (rebuild_grids, check_grid_file)
      mci%negative_weights = negative_weights
      mci%verbose = verbose
    end select
  case default
    call msg_fatal ("Integrator '" &
        // char (integration_method) // "' not implemented")
  end select
end subroutine dispatch_mci
```

22.4.5 Phase-space allocation

Allocate a phase-space object according to the variable `$phs_method`.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_phs
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_phs (phs, global, process_id, mapping_defaults, phs_par)
      class(phs_config_t), allocatable, intent(inout) :: phs
      type(rt_data_t), intent(in) :: global
      type(string_t), intent(in) :: process_id
      type(mapping_defaults_t), intent(in), optional :: mapping_defaults
      type(phs_parameters_t), intent(in), optional :: phs_par
      type(string_t) :: phs_method, phs_file, run_id
      logical :: use_equivalences, vis_channels, fatal_beam_decay
      integer :: u_phs
      logical :: exist
      phs_method = var_list_get_sval (global%var_list, &
          var_str ("$phs_method"))
      phs_file = var_list_get_sval (global%var_list, &
          var_str ("$phs_file"))
      use_equivalences = var_list_get_lval (global%var_list, &
          var_str ("?use_vamp_equivalences"))
      vis_channels = var_list_get_lval (global%var_list, &
          var_str ("?vis_channels"))
      fatal_beam_decay = var_list_get_lval (global%var_list, &
          var_str ("?fatal_beam_decay"))
      run_id = &
          var_list_get_sval (global%var_list, var_str ("$run_id"))
      select case (char (phs_method))
      case ("single")
        allocate (phs_single_config_t :: phs)
        if (vis_channels) then
          call msg_warning ("Visualizing phase space channels not " // &
              "available for method 'single'.")
        end if
      case ("wood", "default")
        allocate (phs_wood_config_t :: phs)
        select type (phs)
        type is (phs_wood_config_t)
          if (phs_file /= "") then
              inquire (file = char (phs_file), exist = exist)
              if (exist) then
```

```
                    call msg_message ("Phase space: reading configuration from '" &
                         // char (phs_file) // "'")
                    u_phs = free_unit ()
                    open (u_phs, file = char (phs_file), &
                         action = "read", status = "old")
                    call phs%set_input (u_phs)
                 else
                    call msg_fatal ("Phase space: configuration file '" &
                         // char (phs_file) // "' not found")
                 end if
              end if
              if (present (phs_par)) &
                   call phs%set_parameters (phs_par)
              if (use_equivalences) &
                   call phs%enable_equivalences ()
              if (present (mapping_defaults)) &
                   call phs%set_mapping_defaults (mapping_defaults)
              phs%vis_channels = vis_channels
              phs%fatal_beam_decay = fatal_beam_decay
              phs%os_data = global%os_data
              phs%run_id = run_id
           end select
        case default
           call msg_fatal ("Phase space: parameterization method '" &
                // char (phs_method) // "' not implemented")
        end select
     end subroutine dispatch_phs
```

22.4.6   Random-number generator

Allocate a random-number generator factory according to the variable `$rng_method`, using the current seed in the global record. We take only the lower 15 bits of the seed, so the actual value fits into a positive 16-bit signed integer.

Since we want to guarantee that all random-number generators in a run are independent, we increment the global seed by one after creating the rng factory. If the user wants to have identical sequences, he can always set the seed manually, before it is used.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_rng_factory
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_rng_factory (rng_factory, global)
      class(rng_factory_t), allocatable, intent(out) :: rng_factory
      type(rt_data_t), intent(inout) :: global
      type(string_t) :: rng_method
      integer :: seed
      character(30) :: buffer
      integer(i16) :: s
      rng_method = var_list_get_sval (global%var_list, var_str ("$rng_method"))
      seed = var_list_get_ival (global%var_list, var_str ("seed"))
      s = mod (seed, 32768)
      select case (char (rng_method))
```

```
      case ("unit_test")
         allocate (rng_test_factory_t :: rng_factory)
         call msg_message ("RNG: Initializing Test random-number generator")
      case ("tao")
         allocate (rng_tao_factory_t :: rng_factory)
         call msg_message ("RNG: Initializing TAO random-number generator")
      case default
         call msg_fatal ("Random-number generator '" &
               // char (rng_method) // "' not implemented")
      end select
      write (buffer, "(I0)")  s
      call msg_message ("RNG: Setting seed for random-number generator to " &
            // trim (buffer))
      call rng_factory%init (s)
      call var_list_set_int (global%var_list, var_str ("seed"), seed + 1, &
            is_known = .true.)
    end subroutine dispatch_rng_factory
```

22.4.7 Structure function configuration data

This data type is a container for transferring structure-function specific data
from the `dispatch_sf_data` to the `dispatch_sf_channels` subroutine.

⟨*Dispatch: public*⟩+≡
```
    public :: sf_prop_t
```

⟨*Dispatch: types*⟩≡
```
    type :: sf_prop_t
       real(default), dimension(2) :: isr_eps = 1
    end type sf_prop_t
```

Allocate a structure-function configuration object according to the `sf_method`
string.

The `sf_prop` object can be used to transfer structure-function specific data
up and to the `dispatch_sf_channels` subroutine below, so they can be used
for particular mappings.

The `global` object is intent(inout) because the PDF status may change
during initialization.

The `pdg_in` array is the array of incoming flavors, corresponding to the up-
stream structure function or the beam array. This will be checked for the struc-
ture function in question and replaced by the outgoing flavors. The `pdg_prc`
array is the array of incoming flavors (beam index, component index) for the
hard process.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_sf_data
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_sf_data (data, sf_method, i_beam, sf_prop, global, &
         pdg_in, pdg_prc)
      class(sf_data_t), allocatable, intent(inout) :: data
      type(string_t), intent(in) :: sf_method
      integer, dimension(:), intent(in) :: i_beam
```

```
type(pdg_array_t), dimension(:), intent(inout) :: pdg_in
type(pdg_array_t), dimension(:,:), intent(in) :: pdg_prc
type(sf_prop_t), intent(inout) :: sf_prop
type(rt_data_t), intent(inout) :: global
type(model_t), pointer :: model
type(pdg_array_t), dimension(:), allocatable :: pdg_out
real(default) :: sqrts, isr_alpha, isr_q_max, isr_mass
integer :: isr_order
logical :: isr_recoil
real(default) :: epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass
logical :: epa_recoil
real(default) :: ewa_x_min, ewa_pt_max, ewa_mass
logical :: ewa_keep_momentum, ewa_keep_energy
type(pdg_array_t), dimension(:), allocatable :: pdg_prc1
integer :: ewa_id
type(string_t) :: pdf_name
type(string_t) :: lhapdf_dir, lhapdf_file
type(string_t), dimension(13) :: lhapdf_photon_sets
integer :: lhapdf_member, lhapdf_photon_scheme
logical :: lhapdf_hoppet_b_matching
class(rng_factory_t), allocatable :: rng_factory
logical :: circe1_photon1, circe1_photon2, circe1_generate
real(default) :: circe1_sqrts, circe1_eps
integer :: circe1_version, circe1_chattiness, &
     circe1_revision
character(6) :: circe1_accelerator
logical :: circe2_polarized
type(string_t) :: circe2_design, circe2_file
logical :: beam_events_warn_eof
type(string_t) :: beam_events_dir, beam_events_file
logical :: escan_normalize
lhapdf_photon_sets = [var_str ("DOG0.LHgrid"), var_str ("DOG1.LHgrid"), &
     var_str ("DGG.LHgrid"), var_str ("LACG.LHgrid"), &
     var_str ("GSG0.LHgrid"), var_str ("GSG1.LHgrid"), &
     var_str ("GSG960.LHgrid"), var_str ("GSG961.LHgrid"), &
     var_str ("GRVG0.LHgrid"), var_str ("GRVG1.LHgrid"), &
     var_str ("ACFGPG.LHgrid"), var_str ("WHITG.LHgrid"), &
     var_str ("SASG.LHgrid")]
model => global%model
sqrts = global%get_sqrts ()
associate (var_list => global%var_list)
  select case (char (sf_method))
  case ("sf_test_0", "sf_test_1")
     allocate (sf_test_data_t :: data)
     select type (data)
     type is (sf_test_data_t)
        select case (char (sf_method))
        case ("sf_test_0");  call data%init (model, pdg_in(i_beam(1)))
        case ("sf_test_1");  call data%init (model, pdg_in(i_beam(1)), &
             mode = 1)
        end select
     end select
  case ("pdf_builtin")
     allocate (pdf_builtin_data_t :: data)
```

```
   select type (data)
   type is (pdf_builtin_data_t)
      pdf_name = &
           var_list_get_sval (var_list, var_str ("$pdf_builtin_set"))
      call data%init (global%pdf_builtin_status, &
           model, pdg_in(i_beam(1)), &
           name = pdf_name, &
           path = global%os_data%pdf_builtin_datapath)
   end select
case ("pdf_builtin_photon")
   call msg_fatal ("Currently, there are no photon PDFs built into WHIZARD,", &
        [var_str ("for the photon content inside a proton or neutron use"), &
         var_str ("the 'lhapdf_photon' structure function.")])
case ("lhapdf")
   allocate (lhapdf_data_t :: data)
   if (pdg_array_get (pdg_in(i_beam(1)), 1) == PHOTON) then
      call msg_fatal ("The 'lhapdf' structure is intended only for protons and", &
           [var_str ("pions, please use 'lhapdf_photon' for photon beams.")])
   end if
   lhapdf_dir = &
        var_list_get_sval (var_list, var_str ("$lhapdf_dir"))
   lhapdf_file = &
        var_list_get_sval (var_list, var_str ("$lhapdf_file"))
   lhapdf_member = &
        var_list_get_ival (var_list, var_str ("lhapdf_member"))
   lhapdf_photon_scheme = &
        var_list_get_ival (var_list, var_str ("lhapdf_photon_scheme"))
   lhapdf_hoppet_b_matching = &
        var_list_get_lval (var_list, var_str ("?lhapdf_hoppet_b_matching"))
   select type (data)
   type is (lhapdf_data_t)
      call data%init &
           (global%lhapdf_status, model, pdg_in(i_beam(1)), &
            lhapdf_dir, lhapdf_file, lhapdf_member, &
            lhapdf_photon_scheme, lhapdf_hoppet_b_matching)
   end select
case ("lhapdf_photon")
   allocate (lhapdf_data_t :: data)
   if (pdg_array_get_length (pdg_in(i_beam(1))) /= 1 .or. &
        pdg_array_get (pdg_in(i_beam(1)), 1) /= PHOTON) then
      call msg_fatal ("The 'lhapdf_photon' structure function is exclusively for", &
           [var_str ("photon PDFs, i.e. for photons as beam particles")])
   end if
   lhapdf_dir = &
        var_list_get_sval (var_list, var_str ("$lhapdf_dir"))
   lhapdf_file = &
        var_list_get_sval (var_list, var_str ("$lhapdf_photon_file"))
   lhapdf_member = &
        var_list_get_ival (var_list, var_str ("lhapdf_member"))
   lhapdf_photon_scheme = &
        var_list_get_ival (var_list, var_str ("lhapdf_photon_scheme"))
   if (.not. any (lhapdf_photon_sets == lhapdf_file)) then
      call msg_fatal ("This PDF set is not supported or not " // &
           "intended for photon beams.")
```

```
      end if
      select type (data)
      type is (lhapdf_data_t)
         call data%init &
              (global%lhapdf_status, model, pdg_in(i_beam(1)), &
               lhapdf_dir, lhapdf_file, lhapdf_member, &
               lhapdf_photon_scheme)
      end select
   case ("isr")
      allocate (isr_data_t :: data)
      isr_alpha = var_list_get_rval (var_list, var_str ("isr_alpha"))
      if (isr_alpha == 0) then
         isr_alpha = (var_list_get_rval (var_list, var_str ("ee"))) &
              ** 2 / (4 * PI)
      end if
      isr_q_max = var_list_get_rval (var_list, var_str ("isr_q_max"))
      if (isr_q_max == 0) then
         isr_q_max = sqrts
      end if
      isr_mass  = var_list_get_rval (var_list, var_str ("isr_mass"))
      isr_order = var_list_get_ival (var_list, var_str ("isr_order"))
      isr_recoil = var_list_get_lval (var_list, var_str ("?isr_recoil"))
      select type (data)
      type is (isr_data_t)
         call data%init &
              (model, pdg_in (i_beam(1)), isr_alpha, isr_q_max, &
               isr_mass, isr_order, isr_recoil)
         call data%check ()
         sf_prop%isr_eps(i_beam(1)) = data%get_eps ()
      end select
   case ("epa")
      allocate (epa_data_t :: data)
      epa_alpha = var_list_get_rval (var_list, var_str ("epa_alpha"))
      if (epa_alpha == 0) then
         epa_alpha = (var_list_get_rval (var_list, var_str ("ee"))) &
              ** 2 / (4 * PI)
      end if
      epa_x_min = var_list_get_rval (var_list, var_str ("epa_x_min"))
      epa_q_min = var_list_get_rval (var_list, var_str ("epa_q_min"))
      epa_e_max = var_list_get_rval (var_list, var_str ("epa_e_max"))
      if (epa_e_max == 0) then
         epa_e_max = sqrts
      end if
      epa_mass  = var_list_get_rval (var_list, var_str ("epa_mass"))
      epa_recoil = var_list_get_lval (var_list, var_str ("?epa_recoil"))
      select type (data)
      type is (epa_data_t)
         call data%init &
              (model, pdg_in (i_beam(1)), epa_alpha, epa_x_min, &
               epa_q_min, epa_e_max, epa_mass, epa_recoil)
         call data%check ()
      end select
   case ("ewa")
      allocate (ewa_data_t :: data)
```

```
      allocate (pdg_prc1 (size (pdg_prc, 2)))
      pdg_prc1 = pdg_prc(i_beam(1),:)
      if (any (pdg_array_get_length (pdg_prc1) /= 1) &
            .or. any (pdg_prc1 /= pdg_prc1(1))) then
         call msg_fatal &
             ("EWA: process incoming particle (W/Z) must be unique")
      end if
      ewa_id = abs (pdg_array_get (pdg_prc1(1), 1))
      ewa_x_min = var_list_get_rval (var_list, var_str ("ewa_x_min"))
      ewa_pt_max = var_list_get_rval (var_list, var_str ("ewa_pt_max"))
      if (ewa_pt_max == 0) then
         ewa_pt_max = sqrts
      end if
      ewa_mass = var_list_get_rval (var_list, var_str ("ewa_mass"))
      ewa_keep_momentum = var_list_get_lval (var_list, &
            var_str ("?ewa_keep_momentum"))
      ewa_keep_energy = var_list_get_lval (var_list, &
                var_str ("?ewa_keep_energy"))
      if (ewa_keep_momentum .and. ewa_keep_energy) &
            call msg_fatal (" EWA cannot conserve both energy " &
               // "and momentum.")
      select type (data)
      type is (ewa_data_t)
         call data%init &
               (model, pdg_in (i_beam(1)), ewa_x_min, &
               ewa_pt_max, sqrts, ewa_keep_momentum, &
               ewa_keep_energy, ewa_mass)
         call data%set_id (ewa_id)
         call data%check ()
      end select
   case ("circe1")
      allocate (circe1_data_t :: data)
      select type (data)
      type is (circe1_data_t)
         circe1_photon1 = &
               var_list_get_lval (var_list, var_str ("?circe1_photon1"))
         circe1_photon2 = &
               var_list_get_lval (var_list, var_str ("?circe1_photon2"))
         circe1_sqrts = &
               var_list_get_rval (var_list, var_str ("circe1_sqrts"))
         circe1_eps = &
               var_list_get_rval (var_list, var_str ("circe1_eps"))
         if (circe1_sqrts <= 0)  circe1_sqrts = sqrts
         circe1_generate = &
               var_list_get_lval (var_list, var_str ("?circe1_generate"))
         circe1_version = &
               var_list_get_ival (var_list, var_str ("circe1_ver"))
         circe1_revision = &
               var_list_get_ival (var_list, var_str ("circe1_rev"))
         circe1_accelerator = &
               char (var_list_get_sval (var_list, var_str ("$circe1_acc")))
         circe1_chattiness = &
               var_list_get_ival (var_list, var_str ("circe1_chat"))
         call data%init (model, pdg_in, circe1_sqrts, circe1_eps, &
```

```fortran
                   [circe1_photon1, circe1_photon2], &
                   circe1_version, circe1_revision, circe1_accelerator, &
                   circe1_chattiness)
              if (circe1_generate) then
                 call msg_message ("Circe1: activating generator mode")
                 call dispatch_rng_factory (rng_factory, global)
                 call data%set_generator_mode (rng_factory)
              end if
           end select
        case ("circe2")
           allocate (circe2_data_t :: data)
           select type (data)
           type is (circe2_data_t)
              circe2_polarized = &
                   var_list_get_lval (var_list, var_str ("?circe2_polarized"))
              circe2_file = &
                   var_list_get_sval (var_list, var_str ("$circe2_file"))
              circe2_design = &
                   var_list_get_sval (var_list, var_str ("$circe2_design"))
              call data%init (global%os_data, model, pdg_in, sqrts, &
                   circe2_polarized, circe2_file, circe2_design)
              call msg_message ("Circe2: activating generator mode")
              call dispatch_rng_factory (rng_factory, global)
              call data%set_generator_mode (rng_factory)
           end select
        case ("beam_events")
           allocate (beam_events_data_t :: data)
           select type (data)
           type is (beam_events_data_t)
              beam_events_dir = global%os_data%whizard_beamsimpath
              beam_events_file = var_list_get_sval (var_list, &
                   var_str ("$beam_events_file"))
              beam_events_warn_eof = var_list_get_lval (var_list, &
                   var_str ("?beam_events_warn_eof"))
              call data%init (model, pdg_in, &
                     beam_events_dir, beam_events_file, beam_events_warn_eof)
           end select
        case ("energy_scan")
           escan_normalize = &
                var_list_get_lval (var_list, var_str ("?energy_scan_normalize"))
           allocate (escan_data_t :: data)
           select type (data)
           type is (escan_data_t)
              if (escan_normalize) then
                 call data%init (model, pdg_in)
              else
                 call data%init (model, pdg_in, sqrts)
              end if
           end select
        case default
           call msg_bug ("Structure function '" &
                // char (sf_method) // "' not implemented yet")
        end select
     end associate
```

2841

```
         if (allocated (data)) then
            allocate (pdg_out (size (pdg_prc, 1)))
            call data%get_pdg_out (pdg_out)
            pdg_in(i_beam) = pdg_out
         end if
   end subroutine dispatch_sf_data
```

This is an auxiliary procedure, used by the beam-structure expansion: tell for a given structure function name, whether it corresponds to a pair spectrum ($n = 2$), a single-particle structure function ($n = 1$), or nothing ($n = 0$). Though `energy_scan` can in principle also be a pair spectrum, it always has only one parameter.

⟨*Dispatch: procedures*⟩+≡
```
   function strfun_mode (name) result (n)
     type(string_t), intent(in) :: name
     integer :: n
     select case (char (name))
     case ("none")
        n = 0
     case ("sf_test_0", "sf_test_1")
        n = 1
     case ("pdf_builtin","pdf_builtin_photon", &
           "lhapdf","lhapdf_photon")
        n = 1
     case ("isr","epa","ewa")
        n = 1
     case ("circe1", "circe2")
        n = 2
     case ("beam_events")
        n = 2
     case ("energy_scan")
        n = 2
     case default
        call msg_bug ("Structure function '" // char (name) &
              // "' not supported yet")
     end select
   end function strfun_mode
```

Dispatch a whole structure-function chain, given beam data and beam structure data.

This could be done generically, but we should look at the specific combination of structure functions in order to select appropriate mappings.

The `beam_structure` argument gets copied because we want to expand it to canonical form (one valid structure-function entry per record) before proceeding further.

The `pdg_prc` argument is the array of incoming flavors. The first index is the beam index, the second one the process component index. Each element is itself a PDG array, notrivial if there is a flavor sum for the incoming state of this component.

The dispatcher is divided in two parts. The first part configures the structure function data themselves. After this, we can configure the phase space for the

2842

elementary process.

⟨*Dispatch: public*⟩+≡

```
    public :: dispatch_sf_config
```

⟨*Dispatch: procedures*⟩+≡

```
    subroutine dispatch_sf_config (sf_config, sf_prop, global, pdg_prc)
      type(sf_config_t), dimension(:), allocatable, intent(out) :: sf_config
      type(sf_prop_t), intent(out) :: sf_prop
      type(rt_data_t), intent(inout) :: global
      type(beam_structure_t) :: beam_structure
      class(sf_data_t), allocatable :: sf_data
      type(pdg_array_t), dimension(:,:), intent(in) :: pdg_prc
      type(string_t), dimension(:), allocatable :: prt_in
      type(pdg_array_t), dimension(:), allocatable :: pdg_in
      type(flavor_t) :: flv_in
      integer :: n_beam, n_record, i
      beam_structure = global%beam_structure
      call beam_structure%expand (strfun_mode)
      n_record = beam_structure%get_n_record ()
      allocate (sf_config (n_record))
      n_beam = beam_structure%get_n_beam ()
      if (n_beam > 0) then
         allocate (prt_in (n_beam), pdg_in (n_beam))
         prt_in = beam_structure%get_prt ()
         do i = 1, n_beam
            call flavor_init (flv_in, prt_in(i), global%model)
            pdg_in(i) = flavor_get_pdg (flv_in)
         end do
      else
         n_beam = size (pdg_prc, 1)
         allocate (pdg_in (n_beam))
         pdg_in = pdg_prc(:,1)
      end if
      do i = 1, n_record
         call dispatch_sf_data (sf_data, &
              beam_structure%get_name (i), &
              beam_structure%get_i_entry (i), &
              sf_prop, global, pdg_in, pdg_prc)
         call sf_config(i)%init (beam_structure%get_i_entry (i), sf_data)
         deallocate (sf_data)
      end do
    end subroutine dispatch_sf_config
```

The second part configures channel mappings, using some conditions from the
phase space configuration. If there are no structure functions, we enable a
default setup with a single (dummy) structure-function channel. Otherwise, we
look at the channel collection that we got from the phase-space configuration
step. Each entry should be translated into an independent structure-function
channel, where typically there is one default entry, which could be mapped using
a standard s-channel mapping if the structure function setup recommends this,
and other entries with s-channel resonances. The latter need to be translated
into global mappings from the structure-function chain.

⟨*Dispatch: public*⟩+≡

```
            public :: dispatch_sf_channels
⟨Dispatch: procedures⟩+≡
      subroutine dispatch_sf_channels (sf_channel, sf_string, sf_prop, coll, global)
        type(sf_channel_t), dimension(:), allocatable, intent(out) :: sf_channel
        type(string_t), intent(out) :: sf_string
        type(sf_prop_t), intent(in) :: sf_prop
        type(phs_channel_collection_t), intent(in) :: coll
        type(rt_data_t), intent(in) :: global
        type(beam_structure_t) :: beam_structure
        class(channel_prop_t), allocatable :: prop
        integer :: n_strfun, n_sf_channel, i
        logical :: sf_allow_s_mapping, circe1_map, circe1_generate
        logical :: s_mapping_enable, endpoint_mapping, power_mapping
        integer, dimension(:), allocatable :: s_mapping, single_mapping
        real(default) :: sqrts, s_mapping_power
        real(default) :: circe1_mapping_slope, endpoint_mapping_slope
        real(default) :: power_mapping_eps
        sqrts = global%get_sqrts ()
        beam_structure = global%beam_structure
        call beam_structure%expand (strfun_mode)
        n_strfun = beam_structure%get_n_record ()
        sf_string = beam_structure%to_string (sf_only = .true.)
        sf_allow_s_mapping = &
             var_list_get_lval (global%var_list, var_str ("?sf_allow_s_mapping"))
        circe1_generate = &
             var_list_get_lval (global%var_list, var_str ("?circe1_generate"))
        circe1_map = &
             var_list_get_lval (global%var_list, var_str ("?circe1_map"))
        circe1_mapping_slope = &
             var_list_get_rval (global%var_list, var_str ("circe1_mapping_slope"))
        s_mapping_enable = .false.
        s_mapping_power = 1
        endpoint_mapping = .false.
        endpoint_mapping_slope = 1
        power_mapping = .false.
        select case (char (sf_string))
        case ("", "[any particles]")
        case ("pdf_builtin, none", &
             "pdf_builtin_photon, none", &
             "none, pdf_builtin", &
             "none, pdf_builtin_photon", &
             "lhapdf, none", &
             "lhapdf_photon, none", &
             "none, lhapdf", &
             "none, lhapdf_photon")
        case ("pdf_builtin, none => none, pdf_builtin", &
              "pdf_builtin, none => none, pdf_builtin_photon", &
              "pdf_builtin_photon, none => none, pdf_builtin", &
              "pdf_builtin_photon, none => none, pdf_builtin_photon", &
              "lhapdf, none => none, lhapdf", &
              "lhapdf, none => none, lhapdf_photon", &
              "lhapdf_photon, none => none, lhapdf", &
              "lhapdf_photon, none => none, lhapdf_photon")
           allocate (s_mapping (2), source = [1, 2])
```

2844

```
      s_mapping_enable = .true.
      s_mapping_power = 2
   case ("pdf_builtin, none => none, pdf_builtin => epa, none => none, epa", &
         "pdf_builtin, none => none, pdf_builtin => ewa, none => none, ewa", &
         "pdf_builtin, none => none, pdf_builtin => ewa, none => none, epa", &
         "pdf_builtin, none => none, pdf_builtin => epa, none => none, ewa")
      allocate (s_mapping (2), source = [1, 2])
      s_mapping_enable = .true.
      s_mapping_power = 2
   case ("isr, none", &
        "none, isr")
      allocate (single_mapping (1), source = [1])
   case ("isr, none => none, isr")
      allocate (s_mapping (2), source = [1, 2])
      power_mapping = .true.
      power_mapping_eps = minval (sf_prop%isr_eps)
   case ("isr, none => none, isr => epa, none => none, epa", &
         "isr, none => none, isr => ewa, none => none, ewa", &
         "isr, none => none, isr => ewa, none => none, epa", &
         "isr, none => none, isr => epa, none => none, ewa")
      allocate (s_mapping (2), source = [1, 2])
      power_mapping = .true.
      power_mapping_eps = minval (sf_prop%isr_eps)
   case ("circe1 => isr, none => none, isr => epa, none => none, epa", &
         "circe1 => isr, none => none, isr => ewa, none => none, ewa", &
         "circe1 => isr, none => none, isr => ewa, none => none, epa", &
         "circe1 => isr, none => none, isr => epa, none => none, ewa")
      if (circe1_generate) then
         allocate (s_mapping (2), source = [2, 3])
      else
         allocate (s_mapping (3), source = [1, 2, 3])
         endpoint_mapping = .true.
         endpoint_mapping_slope = circe1_mapping_slope
      end if
      power_mapping = .true.
      power_mapping_eps = minval (sf_prop%isr_eps)
   case ("pdf_builtin, none => none, isr", &
        "pdf_builtin_photon, none => none, isr", &
        "lhapdf, none => none, isr", &
        "lhapdf_photon, none => none, isr")
      allocate (single_mapping (1), source = [2])
   case ("isr, none => none, pdf_builtin", &
        "isr, none => none, pdf_builtin_photon", &
        "isr, none => none, lhapdf", &
        "isr, none => none, lhapdf_photon")
      allocate (single_mapping (1), source = [1])
   case ("epa, none", &
         "none, epa")
      allocate (single_mapping (1), source = [1])
   case ("epa, none => none, epa")
      allocate (single_mapping (2), source = [1, 2])
   case ("epa, none => none, isr", &
        "isr, none => none, epa", &
        "ewa, none => none, isr", &
```

```
      "isr, none => none, ewa")
   allocate (single_mapping (2), source = [1, 2])
case ("pdf_builtin, none => none, epa", &
     "pdf_builtin_photon, none => none, epa", &
     "lhapdf, none => none, epa", &
     "lhapdf_photon, none => none, epa")
   allocate (single_mapping (1), source = [2])
case ("pdf_builtin, none => none, ewa", &
     "pdf_builtin_photon, none => none, ewa", &
     "lhapdf, none => none, ewa", &
     "lhapdf_photon, none => none, ewa")
   allocate (single_mapping (1), source = [2])
case ("epa, none => none, pdf_builtin", &
     "epa, none => none, pdf_builtin_photon", &
     "epa, none => none, lhapdf", &
     "epa, none => none, lhapdf_photon")
   allocate (single_mapping (1), source = [1])
case ("ewa, none => none, pdf_builtin", &
     "ewa, none => none, pdf_builtin_photon", &
     "ewa, none => none, lhapdf", &
     "ewa, none => none, lhapdf_photon")
   allocate (single_mapping (1), source = [1])
case ("ewa, none", &
      "none, ewa")
   allocate (single_mapping (1), source = [1])
case ("ewa, none => none, ewa")
   allocate (single_mapping (2), source = [1, 2])
case ("energy_scan, none => none, energy_scan")
   allocate (s_mapping (2), source = [1, 2])
case ("sf_test_1, none => none, sf_test_1")
   allocate (s_mapping (2), source = [1, 2])
case ("circe1")
   if (circe1_generate) then
      ! no mapping
   else if (circe1_map) then
      allocate (s_mapping (1), source = [1])
      endpoint_mapping = .true.
      endpoint_mapping_slope = circe1_mapping_slope
   else
      allocate (s_mapping (1), source = [1])
      s_mapping_enable = .true.
   end if
case ("circe1 => isr, none => none, isr")
   if (circe1_generate) then
      allocate (s_mapping (2), source = [2, 3])
   else
      allocate (s_mapping (3), source = [1, 2, 3])
      endpoint_mapping = .true.
      endpoint_mapping_slope = circe1_mapping_slope
   end if
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("circe1 => isr, none", &
     "circe1 => none, isr")
```

2846

```
         allocate (single_mapping (1), source = [2])
case ("circe1 => epa, none => none, epa")
   if (circe1_generate) then
      allocate (single_mapping (2), source = [2, 3])
   else
      call msg_fatal ("Circe/EPA: supported with ?circe1_generate=true &
            &only")
   end if
case ("circe1 => ewa, none => none, ewa")
   if (circe1_generate) then
      allocate (single_mapping (2), source = [2, 3])
   else
      call msg_fatal ("Circe/EWA: supported with ?circe1_generate=true &
            &only")
   end if
case ("circe1 => epa, none", &
      "circe1 => none, epa")
   if (circe1_generate) then
      allocate (single_mapping (1), source = [2])
   else
      call msg_fatal ("Circe/EPA: supported with ?circe1_generate=true &
            &only")
   end if
case ("circe1 => epa, none => none, isr", &
      "circe1 => isr, none => none, epa", &
      "circe1 => ewa, none => none, isr", &
      "circe1 => isr, none => none, ewa")
   if (circe1_generate) then
      allocate (single_mapping (2), source = [2, 3])
   else
      call msg_fatal ("Circe/EPA: supported with ?circe1_generate=true &
            &only")
   end if
case ("circe2")
case ("beam_events")
case ("beam_events => isr, none => none, isr")
   allocate (s_mapping (2), source = [2, 3])
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("beam_events => isr, none", &
      "beam_events => none, isr")
   allocate (single_mapping (1), source = [2])
case ("beam_events => epa, none => none, epa")
   allocate (single_mapping (2), source = [2, 3])
case ("beam_events => epa, none", &
      "beam_events => none, epa", &
      "beam_events => ewa, none", &
      "beam_events => none, ewa")
   allocate (single_mapping (1), source = [2])
case ("beam_events => epa, none => none, isr", &
      "beam_events => isr, none => none, epa", &
      "beam_events => ewa, none => none, isr", &
      "beam_events => isr, none => none, ewa")
   allocate (single_mapping (2), source = [2, 3])
```

```
case ("energy_scan")
case default
   call msg_fatal ("Beam structure: " &
         // char (sf_string) // " not supported")
end select
if (sf_allow_s_mapping .and. coll%n > 0) then
   n_sf_channel = coll%n
   allocate (sf_channel (n_sf_channel))
   do i = 1, n_sf_channel
      call sf_channel(i)%init (n_strfun)
      if (allocated (single_mapping)) then
        call sf_channel(i)%activate_mapping (single_mapping)
      end if
      if (allocated (prop))  deallocate (prop)
      call coll%get_entry (i, prop)
      if (allocated (prop)) then
         if (endpoint_mapping .and. power_mapping) then
            select type (prop)
            type is (resonance_t)
               call sf_channel(i)%set_eir_mapping (s_mapping, &
                     a = endpoint_mapping_slope, eps = power_mapping_eps, &
                     m = prop%mass / sqrts, w = prop%width / sqrts)
            type is (on_shell_t)
               call sf_channel(i)%set_eio_mapping (s_mapping, &
                     a = endpoint_mapping_slope, eps = power_mapping_eps, &
                     m = prop%mass / sqrts)
            end select
         else if (endpoint_mapping) then
            select type (prop)
            type is (resonance_t)
               call sf_channel(i)%set_epr_mapping (s_mapping, &
                     a = endpoint_mapping_slope, &
                     m = prop%mass / sqrts, w = prop%width / sqrts)
            type is (on_shell_t)
               call sf_channel(i)%set_epo_mapping (s_mapping, &
                     a = endpoint_mapping_slope, &
                     m = prop%mass / sqrts)
            end select
         else if (power_mapping) then
            select type (prop)
            type is (resonance_t)
               call sf_channel(i)%set_ipr_mapping (s_mapping, &
                     eps = power_mapping_eps, &
                     m = prop%mass / sqrts, w = prop%width / sqrts)
            type is (on_shell_t)
               call sf_channel(i)%set_ipo_mapping (s_mapping, &
                     eps = power_mapping_eps, &
                     m = prop%mass / sqrts)
            end select
         else if (allocated (s_mapping)) then
            select type (prop)
            type is (resonance_t)
               call sf_channel(i)%set_res_mapping (s_mapping, &
                     m = prop%mass / sqrts, w = prop%width / sqrts)
```

```
                  type is (on_shell_t)
                     call sf_channel(i)%set_os_mapping (s_mapping, &
                          m = prop%mass / sqrts)
                  end select
               else if (allocated (single_mapping)) then
                  select type (prop)
                  type is (resonance_t)
                     call sf_channel(i)%set_res_mapping (single_mapping, &
                          m = prop%mass / sqrts, w = prop%width / sqrts)
                  type is (on_shell_t)
                     call sf_channel(i)%set_os_mapping (single_mapping, &
                          m = prop%mass / sqrts)
                  end select
               end if
            else if (endpoint_mapping .and. power_mapping) then
               call sf_channel(i)%set_ei_mapping (s_mapping, &
                    a = endpoint_mapping_slope, eps = power_mapping_eps)
            else if (endpoint_mapping .and. .not. allocated (single_mapping)) then
               call sf_channel(i)%set_ep_mapping (s_mapping, &
                    a = endpoint_mapping_slope)
            else if (power_mapping .and. .not. allocated (single_mapping)) then
               call sf_channel(i)%set_ip_mapping (s_mapping, &
                    eps = power_mapping_eps)
            else if (s_mapping_enable .and. .not. allocated (single_mapping)) then
               call sf_channel(i)%set_s_mapping (s_mapping, &
                    power = s_mapping_power)
            end if
         end do
      else if (sf_allow_s_mapping) then
         allocate (sf_channel (1))
         call sf_channel(1)%init (n_strfun)
         if (allocated (single_mapping)) then
            call sf_channel(1)%activate_mapping (single_mapping)
         else if (endpoint_mapping .and. power_mapping) then
            call sf_channel(i)%set_ei_mapping (s_mapping, &
                 a = endpoint_mapping_slope, eps = power_mapping_eps)
         else if (endpoint_mapping) then
            call sf_channel(1)%set_ep_mapping (s_mapping, &
                 a = endpoint_mapping_slope)
         else if (power_mapping) then
            call sf_channel(1)%set_ip_mapping (s_mapping, &
                 eps = power_mapping_eps)
         else if (s_mapping_enable) then
            call sf_channel(1)%set_s_mapping (s_mapping, &
                 power = s_mapping_power)
         end if
      else
         allocate (sf_channel (1))
         call sf_channel(1)%init (n_strfun)
         if (allocated (single_mapping)) then
            call sf_channel(1)%activate_mapping (single_mapping)
         end if
      end if
    end subroutine dispatch_sf_channels
```

22.4.8 Event I/O stream

⟨*Dispatch: public*⟩+≡

```
public :: dispatch_eio
```

⟨*Dispatch: procedures*⟩+≡

```
subroutine dispatch_eio (eio, method, global)
  class(eio_t), intent(inout), allocatable :: eio
  type(string_t), intent(in) :: method
  type(rt_data_t), intent(in) :: global
  logical :: check, keep_beams, recover_beams
  logical :: write_sqme_prc, write_sqme_ref, write_sqme_alt
  type(string_t) :: lhef_version, lhef_extension
  type(string_t) :: extension_default, debug_extension, extension_hepmc, &
        extension_lha, extension_hepevt, extension_ascii_short, &
        extension_ascii_long, extension_athena, extension_mokka, &
        extension_stdhep, extension_stdhep_up, extension_raw, &
        extension_hepevt_verb, extension_lha_verb
  integer :: checkpoint
  logical :: show_process, show_transforms, show_decay, verbose, pacify
  select case (char (method))
  case ("raw")
     allocate (eio_raw_t :: eio)
     select type (eio)
     type is (eio_raw_t)
        check = var_list_get_lval &
             (global%var_list, var_str ("?check_event_file"))
        extension_raw = &
             var_list_get_sval (global%var_list, var_str ("$extension_raw"))
        call eio%set_parameters (check, extension_raw)
     end select
  case ("checkpoint")
     allocate (eio_checkpoints_t :: eio)
     select type (eio)
     type is (eio_checkpoints_t)
        checkpoint = var_list_get_ival &
             (global%var_list, var_str ("checkpoint"))
        pacify = var_list_get_lval &
             (global%var_list, var_str ("?pacify"))
        call eio%set_parameters (checkpoint, blank = pacify)
     end select
  case ("lhef")
     allocate (eio_lhef_t :: eio)
     select type (eio)
     type is (eio_lhef_t)
        keep_beams = &
             var_list_get_lval (global%var_list, var_str ("?keep_beams"))
        recover_beams = &
             var_list_get_lval (global%var_list, var_str ("?recover_beams"))
        lhef_version = &
             var_list_get_sval (global%var_list, var_str ("$lhef_version"))
        lhef_extension = &
```

```
               var_list_get_sval (global%var_list, var_str ("$lhef_extension"))
         write_sqme_prc = var_list_get_lval (global%var_list, &
               var_str ("?lhef_write_sqme_prc"))
         write_sqme_ref = var_list_get_lval (global%var_list, &
               var_str ("?lhef_write_sqme_ref"))
         write_sqme_alt = var_list_get_lval (global%var_list, &
               var_str ("?lhef_write_sqme_alt"))
         call eio%set_parameters (keep_beams, recover_beams, &
               char (lhef_version), lhef_extension, &
               write_sqme_ref, write_sqme_prc, write_sqme_alt)
      end select
case ("hepmc")
   allocate (eio_hepmc_t :: eio)
   select type (eio)
   type is (eio_hepmc_t)
      keep_beams = &
            var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_hepmc = var_list_get_sval ( &
            global%var_list, var_str ("$extension_hepmc"))
      call eio%set_parameters (keep_beams, extension_hepmc)
   end select
case ("stdhep")
   allocate (eio_stdhep_hepevt_t :: eio)
   select type (eio)
   type is (eio_stdhep_hepevt_t)
      keep_beams = &
            var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_stdhep = var_list_get_sval ( &
            global%var_list, var_str ("$extension_stdhep"))
      call eio%set_parameters (keep_beams, extension_stdhep)
   end select
case ("stdhep_up")
   allocate (eio_stdhep_hepeup_t :: eio)
   select type (eio)
   type is (eio_stdhep_hepeup_t)
      keep_beams = &
            var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_stdhep_up = var_list_get_sval ( &
            global%var_list, var_str ("$extension_stdhep_up"))
      call eio%set_parameters (keep_beams, extension_stdhep_up)
   end select
case ("ascii")
   allocate (eio_ascii_ascii_t :: eio)
   select type (eio)
   type is (eio_ascii_ascii_t)
      keep_beams = &
            var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_default = var_list_get_sval ( &
            global%var_list, var_str ("$extension_default"))
      call eio%set_parameters (keep_beams, extension_default)
   end select
case ("athena")
   allocate (eio_ascii_athena_t :: eio)
   select type (eio)
```

```
      type is (eio_ascii_athena_t)
         keep_beams = &
              var_list_get_lval (global%var_list, var_str ("?keep_beams"))
         extension_athena = var_list_get_sval ( &
              global%var_list, var_str ("$extension_athena"))
         call eio%set_parameters (keep_beams, extension_athena)
      end select
case ("debug")
   allocate (eio_ascii_debug_t :: eio)
   select type (eio)
   type is (eio_ascii_debug_t)
      keep_beams = &
           var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      debug_extension = var_list_get_sval ( &
           global%var_list, var_str ("$debug_extension"))
      show_process = var_list_get_lval ( &
           global%var_list, var_str ("?debug_process"))
      show_transforms = var_list_get_lval ( &
           global%var_list, var_str ("?debug_transforms"))
      show_decay = var_list_get_lval ( &
           global%var_list, var_str ("?debug_decay"))
      verbose = var_list_get_lval ( &
           global%var_list, var_str ("?debug_verbose"))
      call eio%set_parameters (keep_beams, debug_extension, &
           show_process, show_transforms, show_decay, verbose)
   end select
case ("hepevt")
   allocate (eio_ascii_hepevt_t :: eio)
   select type (eio)
   type is (eio_ascii_hepevt_t)
      keep_beams = &
           var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_hepevt = var_list_get_sval ( &
           global%var_list, var_str ("$extension_hepevt"))
      call eio%set_parameters (keep_beams, extension_hepevt)
   end select
case ("hepevt_verb")
   allocate (eio_ascii_hepevt_verb_t :: eio)
   select type (eio)
   type is (eio_ascii_hepevt_verb_t)
      keep_beams = &
           var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_hepevt_verb = var_list_get_sval ( &
           global%var_list, var_str ("$extension_hepevt_verb"))
      call eio%set_parameters (keep_beams, extension_hepevt_verb)
   end select
case ("lha")
   allocate (eio_ascii_lha_t :: eio)
   select type (eio)
   type is (eio_ascii_lha_t)
      keep_beams = &
           var_list_get_lval (global%var_list, var_str ("?keep_beams"))
      extension_lha = var_list_get_sval ( &
           global%var_list, var_str ("$extension_lha"))
```

```
            call eio%set_parameters (keep_beams, extension_lha)
      end select
   case ("lha_verb")
      allocate (eio_ascii_lha_verb_t :: eio)
      select type (eio)
      type is (eio_ascii_lha_verb_t)
         keep_beams = &
              var_list_get_lval (global%var_list, var_str ("?keep_beams"))
         extension_lha_verb = var_list_get_sval ( &
              global%var_list, var_str ("$extension_lha_verb"))
         call eio%set_parameters (keep_beams, extension_lha_verb)
      end select
   case ("long")
      allocate (eio_ascii_long_t :: eio)
      select type (eio)
      type is (eio_ascii_long_t)
         keep_beams = &
              var_list_get_lval (global%var_list, var_str ("?keep_beams"))
         extension_ascii_long = var_list_get_sval ( &
              global%var_list, var_str ("$extension_ascii_long"))
         call eio%set_parameters (keep_beams, extension_ascii_long)
      end select
   case ("mokka")
      allocate (eio_ascii_mokka_t :: eio)
      select type (eio)
      type is (eio_ascii_mokka_t)
         keep_beams = &
              var_list_get_lval (global%var_list, var_str ("?keep_beams"))
         extension_mokka = var_list_get_sval ( &
              global%var_list, var_str ("$extension_mokka"))
         call eio%set_parameters (keep_beams, extension_mokka)
      end select
   case ("short")
      allocate (eio_ascii_short_t :: eio)
      select type (eio)
      type is (eio_ascii_short_t)
         keep_beams = &
              var_list_get_lval (global%var_list, var_str ("?keep_beams"))
         extension_ascii_short = var_list_get_sval ( &
              global%var_list, var_str ("$extension_ascii_short"))
         call eio%set_parameters (keep_beams, extension_ascii_short)
      end select
   case ("weight_stream")
      allocate (eio_weights_t :: eio)
      select type (eio)
      type is (eio_weights_t)
         pacify = var_list_get_lval &
              (global%var_list, var_str ("?pacify"))
         call eio%set_parameters (pacify = pacify)
      end select
   case default
      call msg_fatal ("Event I/O method '" // char (method) &
           // "' not implemented")
   end select
```

```
      end subroutine dispatch_eio
```

22.4.9 QCD coupling

Allocate the `alpha` (running coupling) component of the `qcd` block with a concrete implementation, depending on the variable settings in the `global` record.

If a fixed α_s is requested, we do not allocate the `qcd%alpha` object. In this case, the matrix element code will just take the model parameter as-is, which implies fixed α_s. If the object is allocated, the α_s value is computed and updated for each matrix-element call.

⟨*Dispatch: public*⟩+≡
```
      public :: dispatch_qcd
```

⟨*Dispatch: procedures*⟩+≡
```
      subroutine dispatch_qcd (qcd, global)
        type(qcd_t), intent(inout) :: qcd
        type(rt_data_t), intent(inout), target :: global
        type(var_list_t), pointer :: var_list
        logical :: fixed, from_mz, from_pdf_builtin, from_lhapdf, from_lambda_qcd
        real(default) :: mz, alpha_val, lambda
        integer :: nf, order, lhapdf_member
        type(string_t) :: pdfset, lhapdf_dir, lhapdf_file
        var_list => global%var_list
        fixed = &
             var_list_get_lval (var_list, var_str ("?alpha_s_is_fixed"))
        from_mz = &
             var_list_get_lval (var_list, var_str ("?alpha_s_from_mz"))
        from_pdf_builtin = &
             var_list_get_lval (var_list, var_str ("?alpha_s_from_pdf_builtin"))
        from_lhapdf = &
             var_list_get_lval (var_list, var_str ("?alpha_s_from_lhapdf"))
        from_lambda_qcd = &
             var_list_get_lval (var_list, var_str ("?alpha_s_from_lambda_qcd"))
        pdfset = &
             var_list_get_sval (var_list, var_str ("$pdf_builtin_set"))
        lambda = &
             var_list_get_rval (var_list, var_str ("lambda_qcd"))
        nf = &
             var_list_get_ival (var_list, var_str ("alpha_s_nf"))
        order = &
             var_list_get_ival (var_list, var_str ("alpha_s_order"))
        lhapdf_dir = &
             var_list_get_sval (var_list, var_str ("$lhapdf_dir"))
        lhapdf_file = &
             var_list_get_sval (var_list, var_str ("$lhapdf_file"))
        lhapdf_member = &
             var_list_get_ival (var_list, var_str ("lhapdf_member"))
        var_list => model_get_var_list_ptr (global%model)
        if (var_list_exists (var_list, var_str ("mZ"))) then
           mz = var_list_get_rval (var_list, var_str ("mZ"))
        else
           mz = MZ_REF
```

```
      end if
      if (var_list_exists (var_list, var_str ("alphas"))) then
         alpha_val = var_list_get_rval (var_list, var_str ("alphas"))
      else
         alpha_val = ALPHA_QCD_MZ_REF
      end if
      if (allocated (qcd%alpha))  deallocate (qcd%alpha)
      if (from_lhapdf .and. from_pdf_builtin) then
          call msg_fatal (" Mixing alphas evolution",  &
               [var_str (" from LHAPDF and builtin PDF is not permitted")])
      end if
      select case (count ([from_mz, from_pdf_builtin, from_lhapdf, from_lambda_qcd]))
      case (0)
         if (fixed) then
            allocate (alpha_qcd_fixed_t :: qcd%alpha)
         else
            call msg_fatal ("QCD alpha: no calculation mode set")
         end if
      case (2:)
         call msg_fatal ("QCD alpha: calculation mode is ambiguous")
      case (1)
         if (fixed) then
            call msg_fatal ("QCD alpha: use '?alpha_s_is_fixed = false' for " // &
                 "running alphas")
         else if (from_mz) then
            allocate (alpha_qcd_from_scale_t :: qcd%alpha)
         else if (from_pdf_builtin) then
            allocate (alpha_qcd_pdf_builtin_t :: qcd%alpha)
         else if (from_lhapdf) then
            allocate (alpha_qcd_lhapdf_t :: qcd%alpha)
         else if (from_lambda_qcd) then
            allocate (alpha_qcd_from_lambda_t :: qcd%alpha)
         end if
         call msg_message ("QCD alpha: using a running strong coupling")
      end select
      select type (alpha => qcd%alpha)
      type is (alpha_qcd_fixed_t)
         alpha%val = alpha_val
      type is (alpha_qcd_from_scale_t)
         alpha%mu_ref = mz
         alpha%ref = alpha_val
         alpha%order = order
         alpha%nf = nf
      type is (alpha_qcd_from_lambda_t)
         alpha%lambda = lambda
         alpha%order = order
         alpha%nf = nf
      type is (alpha_qcd_pdf_builtin_t)
         call alpha%init (global%pdf_builtin_status, pdfset, &
              global%os_data%pdf_builtin_datapath)
      type is (alpha_qcd_lhapdf_t)
         call alpha%init (global%lhapdf_status, lhapdf_file, &
              lhapdf_member, lhapdf_dir)
      end select
```

```
      end subroutine dispatch_qcd
```

22.4.10   Parton Shower

Transfer the settings and parameters for shower and hadronization from the
`global` object to the `shower_settings` object.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_shower
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_shower (shower_settings, global)
      class(shower_settings_t), intent(inout) :: shower_settings
      type(rt_data_t), intent(in), target :: global
      type(var_list_t), pointer :: var_list
      var_list => global%var_list
      call shower_settings%init (var_list)
    end subroutine dispatch_shower
```

22.4.11   Event Transform

An event transform is responsible for dressing a partonic event. Since event
transforms are not mutually exclusive but are concatenated, we provide indi-
vidual dispatchers for each of them.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_evt_decay
    public :: dispatch_evt_shower
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_evt_decay (evt, global)
      class(evt_t), intent(out), pointer :: evt
      type(rt_data_t), intent(in) :: global
      logical :: allow_decays
      allow_decays = &
           var_list_get_lval (global%var_list, var_str ("?allow_decays"))
      if (allow_decays) then
         allocate (evt_decay_t :: evt)
         call msg_message ("Simulate: activating decays")
      else
         evt => null ()
      end if
    end subroutine dispatch_evt_decay

    subroutine dispatch_evt_shower (evt, global, process)
      class(evt_t), intent(out), pointer :: evt
      type(rt_data_t), intent(in) :: global
      type(process_t), intent(in), optional, target :: process
      logical :: allow_shower
      type(shower_settings_t) :: settings
      allow_shower = &
           var_list_get_lval (global%var_list, var_str ("?allow_shower"))
      if (allow_shower) then
```

```
        allocate (evt_shower_t :: evt)
        call msg_message ("Simulate: activating parton shower")
        call dispatch_shower (settings, global)
        if (settings%mlm_matching) &
             call msg_message ("Simulate: applying MLM matching")
        if (settings%ckkw_matching) &
             call msg_warning ("Simulate: CKKW(-L) matching not yet supported")
        if (settings%hadronization_active) &
             call msg_message ("Simulate: applying hadronization")
        select type (evt)
        type is (evt_shower_t)
           call evt%init (settings, global%os_data)
           if (present (process)) &
                call evt%setup_pdf (process, global%beam_structure)
        end select
      else
         evt => null ()
      end if
    end subroutine dispatch_evt_shower
```

22.4.12 SLHA Interface

This is a container for the settings for the SLHA Interface

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_slha
```

⟨*Dispatch: procedures*⟩+≡
```
    subroutine dispatch_slha (global, input, spectrum, decays)
      type(rt_data_t), intent(inout), target :: global
      logical, intent(out) :: input, spectrum, decays
      input = var_list_get_lval (global%var_list, &
           var_str ("?slha_read_input"))
      spectrum = var_list_get_lval (global%var_list, &
           var_str ("?slha_read_spectrum"))
      decays = var_list_get_lval (global%var_list, &
           var_str ("?slha_read_decays"))
    end subroutine dispatch_slha
```

22.4.13 Test

This is the master for calling self-test procedures.

⟨*Dispatch: public*⟩+≡
```
    public :: dispatch_test
```

⟨*Dispatch: tests*⟩≡
```
    subroutine dispatch_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Dispatch: execute tests*⟩
```
    end subroutine dispatch_test
```

Select type: process definition

⟨*Dispatch: execute tests*⟩≡
```
call test (dispatch_1, "dispatch_1", &
     "process configuration method", &
     u, results)
```

⟨*Dispatch: tests*⟩+≡
```
subroutine dispatch_1 (u)
  integer, intent(in) :: u
  type(string_t), dimension(2) :: prt_in, prt_out
  type(rt_data_t), target :: global
  class(prc_core_def_t), allocatable :: core_def

  write (u, "(A)")  "* Test output: dispatch_1"
  write (u, "(A)")  "*   Purpose: select process configuration method"
  write (u, "(A)")

  call global%global_init ()
  call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
       .false., is_known = .true.)

  prt_in = [var_str ("a"), var_str ("b")]
  prt_out = [var_str ("c"), var_str ("d")]

  write (u, "(A)")  "* Allocate core_def as prc_test_def"

  call var_list_set_string (global%var_list, var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call dispatch_core_def (core_def, prt_in, prt_out, global)
  select type (core_def)
  type is (prc_test_def_t)
     call core_def%write (u)
  end select

  deallocate (core_def)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate core_def as omega_def"
  write (u, "(A)")

  call var_list_set_string (global%var_list, var_str ("$method"), &
       var_str ("omega"), is_known = .true.)
  call dispatch_core_def (core_def, prt_in, prt_out, global)
  select type (core_def)
  type is (omega_def_t)
     call core_def%write (u)
  end select

  call global%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_1"

end subroutine dispatch_1
```

Select type: process core

⟨*Dispatch: execute tests*⟩+≡
```
    call test (dispatch_2, "dispatch_2", &
        "process core", &
        u, results)
```

⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_2 (u)
      integer, intent(in) :: u
      type(string_t), dimension(2) :: prt_in, prt_out
      type(rt_data_t), target :: global
      class(prc_core_def_t), allocatable :: core_def
      class(prc_core_t), allocatable :: core

      write (u, "(A)")  "* Test output: dispatch_2"
      write (u, "(A)")  "*   Purpose: select process configuration method"
      write (u, "(A)")  "              and allocate process core"
      write (u, "(A)")

      call syntax_model_file_init ()
      call global%global_init ()

      prt_in = [var_str ("a"), var_str ("b")]
      prt_out = [var_str ("c"), var_str ("d")]

      write (u, "(A)")  "* Allocate core as test_t"
      write (u, "(A)")

      call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known = .true.)
      call dispatch_core_def (core_def, prt_in, prt_out, global)
      call dispatch_core (core, core_def)
      select type (core)
      type is (test_t)
         call core%write (u)
      end select

      deallocate (core)
      deallocate (core_def)

      write (u, "(A)")
      write (u, "(A)")  "* Allocate core as prc_omega_t"
      write (u, "(A)")

      call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("omega"), is_known = .true.)
      call dispatch_core_def (core_def, prt_in, prt_out, global)

      call global%read_model (var_str ("Test"), var_str ("Test.mdl"))

      call var_list_set_log (global%var_list, &
```

```
                var_str ("?helicity_selection_active"), &
                .true., is_known = .true.)
        call var_list_set_real (global%var_list, &
                var_str ("helicity_selection_threshold"), &
                1e9_default, is_known = .true.)
        call var_list_set_int (global%var_list, &
                var_str ("helicity_selection_cutoff"), &
                10, is_known = .true.)

        call dispatch_core (core, core_def, &
                global%model, global%get_helicity_selection ())
        call core_def%allocate_driver (core%driver, var_str (""))

        select type (core)
        type is (prc_omega_t)
           call core%write (u)
        end select

        call global%final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")   "* Test output end: dispatch_2"

     end subroutine dispatch_2
```

Select type: integrator core

⟨*Dispatch: execute tests*⟩+≡
```
        call test (dispatch_3, "dispatch_3", &
                "integration method", &
                u, results)
```
⟨*Dispatch: tests*⟩+≡
```
        subroutine dispatch_3 (u)
          integer, intent(in) :: u
          type(rt_data_t), target :: global
          class(mci_t), allocatable :: mci
          type(string_t) :: process_id

          write (u, "(A)")   "* Test output: dispatch_3"
          write (u, "(A)")   "*   Purpose: select integration method"
          write (u, "(A)")

          call global%global_init ()
          process_id = "dispatch_3"

          write (u, "(A)")   "* Allocate MCI as midpoint_t"
          write (u, "(A)")

          call var_list_set_string (global%var_list, &
                var_str ("$integration_method"), &
                var_str ("midpoint"), is_known = .true.)
```

```
call dispatch_mci (mci, global, process_id)
select type (mci)
type is (mci_midpoint_t)
   call mci%write (u)
end select

call mci%final ()
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Allocate MCI as vamp_t"
write (u, "(A)")

call var_list_set_string (global%var_list, &
     var_str ("$integration_method"), &
     var_str ("vamp"), is_known = .true.)
call var_list_set_int (global%var_list, var_str ("threshold_calls"), &
     1, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("min_calls_per_channel"), &
     2, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("min_calls_per_bin"), &
     3, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("min_bins"), &
     4, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("max_bins"), &
     5, is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?stratified"), &
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .false., is_known = .true.)
call var_list_set_real (global%var_list, var_str ("channel_weights_power"),&
     4._default, is_known = .true.)
call var_list_set_log (global%var_list, &
     var_str ("?vamp_history_global_verbose"), &
     .true., is_known = .true.)
call var_list_set_log (global%var_list, &
     var_str ("?vamp_history_channels"), &
     .true., is_known = .true.)
call var_list_set_log (global%var_list, &
     var_str ("?vamp_history_channels_verbose"), &
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?stratified"), &
     .false., is_known = .true.)

call dispatch_mci (mci, global, process_id)
select type (mci)
type is (mci_vamp_t)
   call mci%write (u)
   call mci%write_history_parameters (u)
end select

call mci%final ()
deallocate (mci)
```

```
      write (u, "(A)")
      write (u, "(A)")   "* Allocate MCI as vamp_t, allow for negative weights"
      write (u, "(A)")

      call var_list_set_string (global%var_list, &
           var_str ("$integration_method"), &
           var_str ("vamp"), is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?negative_weights"), &
           .true., is_known = .true.)

      call dispatch_mci (mci, global, process_id)
      select type (mci)
      type is (mci_vamp_t)
         call mci%write (u)
         call mci%write_history_parameters (u)
      end select

      call mci%final ()
      deallocate (mci)

      call global%final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: dispatch_3"

    end subroutine dispatch_3
```

Select type: phase-space configuration object

⟨*Dispatch: execute tests*⟩+≡
```
      call test (dispatch_4, "dispatch_4", &
           "phase-space configuration", &
           u, results)
```
⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_4 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      class(phs_config_t), allocatable :: phs
      type(phs_parameters_t) :: phs_par
      type(mapping_defaults_t) :: mapping_defs

      write (u, "(A)")   "* Test output: dispatch_4"
      write (u, "(A)")   "*   Purpose: select phase-space configuration method"
      write (u, "(A)")

      call global%global_init ()

      write (u, "(A)")   "* Allocate PHS as phs_single_t"
      write (u, "(A)")

      call var_list_set_string (global%var_list, &
           var_str ("$phs_method"), &
```

```
        var_str ("single"), is_known = .true.)
   call dispatch_phs (phs, global, var_str ("dispatch_4"))
   call phs%write (u)

   call phs%final ()
   deallocate (phs)

   write (u, "(A)")
   write (u, "(A)")  "* Allocate PHS as phs_wood_t"
   write (u, "(A)")

   call var_list_set_string (global%var_list, &
        var_str ("$phs_method"), &
        var_str ("wood"), is_known = .true.)
   call dispatch_phs (phs, global, var_str ("dispatch_4"))
   call phs%write (u)

   call phs%final ()
   deallocate (phs)

   write (u, "(A)")
   write (u, "(A)")  "* Setting parameters for phs_wood_t"
   write (u, "(A)")

   phs_par%m_threshold_s = 123
   phs_par%m_threshold_t = 456
   phs_par%t_channel = 42
   phs_par%off_shell = 17
   phs_par%keep_nonresonant = .false.
   mapping_defs%energy_scale = 987
   mapping_defs%invariant_mass_scale = 654
   mapping_defs%momentum_transfer_scale = 321
   mapping_defs%step_mapping = .false.
   mapping_defs%step_mapping_exp = .false.
   mapping_defs%enable_s_mapping = .true.
   call dispatch_phs (phs, global, var_str ("dispatch_4"), &
        mapping_defs, phs_par)
   call phs%write (u)

   call phs%final ()

   call global%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: dispatch_4"

 end subroutine dispatch_4
```

Select type: random number generator

⟨*Dispatch: execute tests*⟩+≡
```
   call test (dispatch_5, "dispatch_5", &
        "random-number generator", &
```

```
            u, results)
```

```
    subroutine dispatch_5 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      class(rng_factory_t), allocatable :: rng_factory

      write (u, "(A)")  "* Test output: dispatch_5"
      write (u, "(A)")  "*   Purpose: select random-number generator"
      write (u, "(A)")

      call global%global_init ()

      write (u, "(A)")  "* Allocate RNG factory as rng_test_factory_t"
      write (u, "(A)")

      call var_list_set_string (global%var_list, &
           var_str ("$rng_method"), &
           var_str ("unit_test"), is_known = .true.)
      call var_list_set_int (global%var_list, &
           var_str ("seed"), 1, is_known = .true.)
      call dispatch_rng_factory (rng_factory, global)
      call rng_factory%write (u)
      deallocate (rng_factory)

      write (u, "(A)")
      write (u, "(A)")  "* Allocate RNG factory as rng_tao_factory_t"
      write (u, "(A)")

      call var_list_set_string (global%var_list, &
           var_str ("$rng_method"), &
           var_str ("tao"), is_known = .true.)
      call dispatch_rng_factory (rng_factory, global)
      call rng_factory%write (u)
      deallocate (rng_factory)

      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: dispatch_5"

    end subroutine dispatch_5
```

Phase-space configuration with file

```
    call test (dispatch_6, "dispatch_6", &
         "configure phase space using file", &
         u, results)
```

```
    subroutine dispatch_6 (u)
      integer, intent(in) :: u
```

```
type(rt_data_t), target :: global
type(os_data_t) :: os_data
type(process_constants_t) :: process_data
class(phs_config_t), allocatable :: phs
integer :: u_phs

write (u, "(A)")  "* Test output: dispatch_6"
write (u, "(A)")  "*   Purpose: select 'wood' phase-space &
    &for a test process"
write (u, "(A)")  "*           and read phs configuration from file"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process"
write (u, "(A)")

call global%global_init ()

call os_data_init (os_data)
call syntax_model_file_init ()
call global%read_model (var_str ("Test"), var_str ("Test.mdl"))

call syntax_phs_forest_init ()

call init_test_process_data (var_str ("dispatch_6"), process_data)

write (u, "(A)")  "* Write phase-space file"

u_phs = free_unit ()
open (u_phs, file = "dispatch_6.phs", action = "write", status = "replace")
call write_test_phs_file (u_phs, var_str ("dispatch_6"))
close (u_phs)

write (u, "(A)")
write (u, "(A)")  "* Allocate PHS as phs_wood_t"
write (u, "(A)")

call var_list_set_string (global%var_list, &
    var_str ("$phs_method"), &
    var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, &
    var_str ("$phs_file"), &
    var_str ("dispatch_6.phs"), is_known = .true.)
call dispatch_phs (phs, global, var_str ("dispatch_6"))

call phs%init (process_data, global%model)
call phs%configure (sqrts = 1000._default)

call phs%write (u)
write (u, "(A)")
select type (phs)
type is (phs_wood_config_t)
   call phs%write_forest (u)
end select
```

```
      call phs%final ()

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: dispatch_6"

    end subroutine dispatch_6
```

Select type: structure-function data

⟨*Dispatch: execute tests*⟩+≡
```
      call test (dispatch_7, "dispatch_7", &
          "structure-function data", &
          u, results)
```
⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_7 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      type(os_data_t) :: os_data
      type(string_t) :: prt, sf_method
      type(sf_prop_t) :: sf_prop
      class(sf_data_t), allocatable :: data
      type(pdg_array_t), dimension(1) :: pdg_in
      type(pdg_array_t), dimension(1,1) :: pdg_prc
      type(pdg_array_t), dimension(1) :: pdg_out
      integer, dimension(:), allocatable :: pdg1

      write (u, "(A)")  "* Test output: dispatch_7"
      write (u, "(A)")  "*   Purpose: select and configure &
          &structure function data"
      write (u, "(A)")

      call global%global_init ()

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call global%read_model (var_str ("QCD"), var_str ("QCD.mdl"))

      call reset_interaction_counter ()
      call var_list_set_real (global%var_list, var_str ("sqrts"), &
          14000._default, is_known = .true.)
      prt = "p"
      call global%beam_structure%init_sf ([prt, prt], [1])
      pdg_in = 2212

      write (u, "(1x,A,L1)")  "Initialized = ", &
          pdf_builtin_status_is_initialized (global%pdf_builtin_status)
      write (u, "(A)")

      write (u, "(A)")  "* Allocate data as sf_pdf_builtin_t"
```

2866

```
    write (u, "(A)")

    sf_method = "pdf_builtin"
    call dispatch_sf_data &
         (data, sf_method, [1], sf_prop, global, pdg_in, pdg_prc)
    call data%write (u)

    write (u, "(A)")
    write (u, "(1x,A,L1)")  "Initialized = ", &
         pdf_builtin_status_is_initialized (global%pdf_builtin_status)
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(A)")
    write (u, "(1x,A,99(1x,I0))")  "PDG(out) = ", pdg1

    deallocate (data)

    write (u, "(A)")
    write (u, "(A)")  "* Reset and allocate data for different PDF set"
    write (u, "(A)")

    call pdf_builtin_status_reset (global%pdf_builtin_status)
    pdg_in = 2212

    write (u, "(1x,A,L1)")  "Initialized = ", &
         pdf_builtin_status_is_initialized (global%pdf_builtin_status)
    write (u, "(A)")

    call var_list_set_string (global%var_list, var_str ("$pdf_builtin_set"), &
         var_str ("CTEQ6M"), is_known = .true.)
    sf_method = "pdf_builtin"
    call dispatch_sf_data &
         (data, sf_method, [1], sf_prop, global, pdg_in, pdg_prc)
    call data%write (u)

    write (u, "(A)")
    write (u, "(1x,A,L1)")  "Initialized = ", &
         pdf_builtin_status_is_initialized (global%pdf_builtin_status)
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(A)")
    write (u, "(1x,A,99(1x,I0))")  "PDG(out) = ", pdg1

    deallocate (data)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_7"

  end subroutine dispatch_7
```

Beam structure

⟨*Dispatch: execute tests*⟩+≡
```
    call test (dispatch_8, "dispatch_8", &
         "beam structure", &
         u, results)
```
⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_8 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      type(os_data_t) :: os_data
      type(flavor_t), dimension(2) :: flv
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_prop_t) :: sf_prop
      type(sf_channel_t), dimension(:), allocatable :: sf_channel
      type(phs_channel_collection_t) :: coll
      type(string_t) :: sf_string
      integer :: i
      type(pdg_array_t), dimension (2,1) :: pdg_prc

      write (u, "(A)")  "* Test output: dispatch_8"
      write (u, "(A)")  "*   Purpose: configure a structure-function chain"
      write (u, "(A)")

      call global%global_init ()

      call os_data_init (os_data)
      call syntax_model_file_init ()
      call global%read_model (var_str ("QCD"), var_str ("QCD.mdl"))

      write (u, "(A)")  "* Allocate LHC beams with PDF builtin"
      write (u, "(A)")

      call flavor_init (flv(1), PROTON, global%model)
      call flavor_init (flv(2), PROTON, global%model)

      call reset_interaction_counter ()
      call var_list_set_real (global%var_list, var_str ("sqrts"), &
           14000._default, is_known = .true.)

      call global%beam_structure%init_sf (flavor_get_name (flv), [1])
      call global%beam_structure%set_sf (1, 1, var_str ("pdf_builtin"))

      call dispatch_sf_config (sf_config, sf_prop, global, pdg_prc)
      do i = 1, size (sf_config)
         call sf_config(i)%write (u)
      end do

      call dispatch_sf_channels (sf_channel, sf_string, sf_prop, coll, global)
      write (u, "(1x,A)")  "Mapping configuration:"
      do i = 1, size (sf_channel)
         write (u, "(2x)", advance = "no")
         call sf_channel(i)%write (u)
      end do
```

2868

```
      write (u, "(A)")
      write (u, "(A)")   "* Allocate ILC beams with CIRCE1"
      write (u, "(A)")

      call global%read_model (var_str ("QED"), var_str ("QED.mdl"))
      call flavor_init (flv(1), ELECTRON, global%model)
      call flavor_init (flv(2),-ELECTRON, global%model)

      call reset_interaction_counter ()
      call var_list_set_real (global%var_list, var_str ("sqrts"), &
           500._default, is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?circe1_generate"), &
           .false., is_known = .true.)

      call global%beam_structure%init_sf (flavor_get_name (flv), [1])
      call global%beam_structure%set_sf (1, 1, var_str ("circe1"))

      call dispatch_sf_config (sf_config, sf_prop, global, pdg_prc)
      do i = 1, size (sf_config)
         call sf_config(i)%write (u)
      end do

      call dispatch_sf_channels (sf_channel, sf_string, sf_prop, coll, global)
      write (u, "(1x,A)")   "Mapping configuration:"
      do i = 1, size (sf_channel)
         write (u, "(2x)", advance = "no")
         call sf_channel(i)%write (u)
      end do

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: dispatch_8"

    end subroutine dispatch_8
```

Event I/O

⟨*Dispatch: execute tests*⟩+≡
```
    call test (dispatch_9, "dispatch_9", &
         "event I/O", &
         u, results)
```

⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_9 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      class(eio_t), allocatable :: eio
```

```
write (u, "(A)")  "* Test output: dispatch_9"
write (u, "(A)")  "*   Purpose: allocate an event I/O (eio) stream"
write (u, "(A)")

call global%global_init ()

write (u, "(A)")  "* Allocate as raw"
write (u, "(A)")

call dispatch_eio (eio, var_str ("raw"), global)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as checkpoints:"
write (u, "(A)")

call dispatch_eio (eio, var_str ("checkpoint"), global)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as LHEF:"
write (u, "(A)")

call var_list_set_string (global%var_list, var_str ("$lhef_extension"), &
     var_str ("lhe_custom"), is_known = .true.)
call dispatch_eio (eio, var_str ("lhef"), global)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as HepMC:"
write (u, "(A)")

call dispatch_eio (eio, var_str ("hepmc"), global)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as weight_stream"
```

```
      write (u, "(A)")

      call dispatch_eio (eio, var_str ("weight_stream"), global)

      call eio%write (u)

      call eio%final ()
      deallocate (eio)

      write (u, "(A)")
      write (u, "(A)")  "* Allocate as debug format"
      write (u, "(A)")

      call var_list_set_log (global%var_list, var_str ("?debug_verbose"), &
           .false., is_known = .true.)
      call dispatch_eio (eio, var_str ("debug"), global)

      call eio%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call eio%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: dispatch_9"

    end subroutine dispatch_9
```

Update process core parameters

This test dispatches a process core, temporarily modifies parameters, then restores the original.

⟨*Dispatch: execute tests*⟩+≡
```
    call test (dispatch_10, "dispatch_10", &
         "process core update", &
         u, results)
```

⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_10 (u)
      integer, intent(in) :: u
      type(string_t), dimension(2) :: prt_in, prt_out
      type(rt_data_t), target :: global
      class(prc_core_def_t), allocatable :: core_def
      class(prc_core_t), allocatable :: core, saved_core
      type(var_list_t), pointer :: var_list

      write (u, "(A)")  "* Test output: dispatch_10"
      write (u, "(A)")  "*   Purpose: select process configuration method,"
      write (u, "(A)")  "             allocate process core,"
      write (u, "(A)")  "             temporarily reset parameters"
      write (u, "(A)")
```

```
call syntax_model_file_init ()
call global%global_init ()

prt_in = [var_str ("a"), var_str ("b")]
prt_out = [var_str ("c"), var_str ("d")]

write (u, "(A)")  "* Allocate core as prc_omega_t"
write (u, "(A)")

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("omega"), is_known = .true.)
call dispatch_core_def (core_def, prt_in, prt_out, global)

call global%read_model (var_str ("Test"), var_str ("Test.mdl"))

call dispatch_core (core, core_def, global%model)
call core_def%allocate_driver (core%driver, var_str (""))

select type (core)
type is (prc_omega_t)
   call core%write (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Update core with modified model and helicity selection"
write (u, "(A)")

var_list => model_get_var_list_ptr (global%model)
call var_list_set_real (var_list, var_str ("gy"), 2._default, &
     is_known = .true.)
call model_parameters_update (global%model)

call var_list_set_log (global%var_list, &
     var_str ("?helicity_selection_active"), &
     .true., is_known = .true.)
call var_list_set_real (global%var_list, &
     var_str ("helicity_selection_threshold"), &
     2e10_default, is_known = .true.)
call var_list_set_int (global%var_list, &
     var_str ("helicity_selection_cutoff"), &
     5, is_known = .true.)

call dispatch_core_update (core, global%model, &
     global%get_helicity_selection (), &
     saved_core = saved_core)
select type (core)
type is (prc_omega_t)
   call core%write (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Restore core from save"
write (u, "(A)")
```

```
    call dispatch_core_restore (core, saved_core)
    select type (core)
    type is (prc_omega_t)
       call core%write (u)
    end select

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_10"

  end subroutine dispatch_10
```

QCD Coupling

This test dispatches an `qcd` object, which is used to compute the (running) coupling by one of several possible methods.

⟨*Dispatch: execute tests*⟩+≡
```
    call test (dispatch_11, "dispatch_11", &
         "QCD coupling", &
         u, results)
```

⟨*Dispatch: tests*⟩+≡
```
    subroutine dispatch_11 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      type(qcd_t) :: qcd
      type(var_list_t), pointer :: model_vars

      write (u, "(A)")  "* Test output: dispatch_11"
      write (u, "(A)")  "*   Purpose: select QCD coupling formula"
      write (u, "(A)")

      call syntax_model_file_init ()
      call global%global_init ()
      call global%read_model (var_str ("SM"), var_str ("SM.mdl"))
      model_vars => model_get_var_list_ptr (global%model)

      write (u, "(A)")  "* Allocate alpha_s as fixed"
      write (u, "(A)")

      call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
           .true., is_known = .true.)
      call dispatch_qcd (qcd, global)
      call qcd%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Allocate alpha_s as running (built-in)"
      write (u, "(A)")

      call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
```

```
        .false., is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?alpha_s_from_mz"), &
        .true., is_known = .true.)
    call var_list_set_int &
        (global%var_list, var_str ("alpha_s_order"), 1, is_known = .true.)
    call var_list_set_real &
        (model_vars, var_str ("alphas"), 0.1234_default, &
         is_known=.true.)
    call var_list_set_real &
        (model_vars, var_str ("mZ"), 91.234_default, &
         is_known=.true.)
    call dispatch_qcd (qcd, global)
    call qcd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate alpha_s as running (built-in, Lambda defined)"
    write (u, "(A)")

    call var_list_set_log (global%var_list, var_str ("?alpha_s_from_mz"), &
        .false., is_known = .true.)
    call var_list_set_log (global%var_list, &
        var_str ("?alpha_s_from_lambda_qcd"), &
        .true., is_known = .true.)
    call var_list_set_real &
        (global%var_list, var_str ("lambda_qcd"), 250.e-3_default, &
         is_known=.true.)
    call var_list_set_int &
        (global%var_list, var_str ("alpha_s_order"), 2, is_known = .true.)
    call var_list_set_int &
        (global%var_list, var_str ("alpha_s_nf"), 4, is_known = .true.)
    call dispatch_qcd (qcd, global)
    call qcd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate alpha_s as running (using builtin PDF set)"
    write (u, "(A)")

    call var_list_set_log (global%var_list, &
        var_str ("?alpha_s_from_lambda_qcd"), &
        .false., is_known = .true.)
    call var_list_set_log &
        (global%var_list, var_str ("?alpha_s_from_pdf_builtin"), &
        .true., is_known = .true.)
    call dispatch_qcd (qcd, global)
    call qcd%write (u)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_11"

  end subroutine dispatch_11
```

Shower settings

This test dispatches an `shower_settings` object, which is used to steer the initial and final state.

⟨*Dispatch: execute tests*⟩+≡
```
call test (dispatch_12, "dispatch_12", &
     "Shower settings", &
     u, results)
```

⟨*Dispatch: tests*⟩+≡
```
subroutine dispatch_12 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: global
  type(shower_settings_t) :: shower_settings

  write (u, "(A)")  "* Test output: dispatch_12"
  write (u, "(A)")  "*   Purpose: setting ISR/FSR shower"
  write (u, "(A)")

  write (u, "(A)")  "* Default settings"
  write (u, "(A)")

  call global%global_init ()
  call var_list_set_log (global%var_list, var_str ("?alpha_s_is_fixed"), &
       .true., is_known = .true.)
  call dispatch_shower (shower_settings, global)
  call write_separator (u)
  call shower_settings%write (u)
  call write_separator (u)

  write (u, "(A)")
  write (u, "(A)")  "* Switch on ISR/FSR showers, hadronization"
  write (u, "(A)")  "      and MLM matching"
  write (u, "(A)")

  call var_list_set_log (global%var_list, var_str ("?ps_fsr_active"), &
       .true., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?ps_isr_active"), &
       .true., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?hadronization_active"), &
       .true., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?mlm_matching"), &
       .true., is_known = .true.)
  call var_list_set_int &
       (global%var_list, var_str ("ps_max_n_flavors"), 4, is_known = .true.)
  call var_list_set_real &
       (global%var_list, var_str ("ps_isr_z_cutoff"), 0.1234_default, &
        is_known=.true.)
  call var_list_set_real (global%var_list, &
       var_str ("mlm_etamax"), 3.456_default, is_known=.true.)
  call var_list_set_string (global%var_list, &
       var_str ("$ps_PYTHIA_PYGIVE"), var_str ("abcdefgh"), is_known=.true.)
  call dispatch_shower (shower_settings, global)
  call write_separator (u)
  call shower_settings%write (u)
```

2875

```
      call write_separator (u)

      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: dispatch_12"

   end subroutine dispatch_12
```

Event transforms

This test dispatches an `evt` (event transform) object. Currently, the only non-trivial transform is the partonic decay chain..

⟨*Dispatch: execute tests*⟩+≡
```
      call test (dispatch_13, "dispatch_13", &
          "event transforms", &
          u, results)
```

⟨*Dispatch: tests*⟩+≡
```
   subroutine dispatch_13 (u)
     integer, intent(in) :: u
     type(rt_data_t), target :: global
     class(evt_t), pointer :: evt

     write (u, "(A)")  "* Test output: dispatch_13"
     write (u, "(A)")  "*   Purpose: configure event transform"
     write (u, "(A)")

     call syntax_model_file_init ()
     call global%global_init ()

     write (u, "(A)")  "* Partonic decays"
     write (u, "(A)")

     call dispatch_evt_decay (evt, global)
     select type (evt)
     type is (evt_decay_t)
        call evt%write (u, show_decay_tree = .true., verbose = .true.)
     end select

     call evt%final ()
     deallocate (evt)

     write (u, "(A)")
     write (u, "(A)")  "* Shower"
     write (u, "(A)")

     call var_list_set_log (global%var_list, var_str ("?allow_shower"), .true., &
         is_known = .true.)
     call dispatch_evt_shower (evt, global)
     select type (evt)
     type is (evt_shower_t)
        call evt%write (u)
```

```
         call write_separator_double (u)
      end select

      call evt%final ()
      deallocate (evt)

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: dispatch_13"

   end subroutine dispatch_13
```

SLHA interface

This rather trivial sets all input values for the SLHA interface to `false`.

⟨*Dispatch: execute tests*⟩+≡
```
      call test (dispatch_14, "dispatch_14", &
           "SLHA interface", &
           u, results)
```
⟨*Dispatch: tests*⟩+≡
```
   subroutine dispatch_14 (u)
     integer, intent(in) :: u
     type(rt_data_t), target :: global
     logical :: input, spectrum, decays

     write (u, "(A)")  "* Test output: dispatch_14"
     write (u, "(A)")  "*   Purpose: SLHA interface settings"
     write (u, "(A)")

     write (u, "(A)")  "* Default settings"
     write (u, "(A)")

     call global%global_init ()
     call dispatch_slha (global, &
          input = input, spectrum = spectrum, decays = decays)

     write (u, "(A,1x,L1)")  " slha_read_input    =", input
     write (u, "(A,1x,L1)")  " slha_read_spectrum =", spectrum
     write (u, "(A,1x,L1)")  " slha_read_decays   =", decays

     call global%final ()
     call global%global_init ()

     write (u, "(A)")
     write (u, "(A)")  "* Set all entries to [false]"
     write (u, "(A)")

     call var_list_set_log (global%var_list, var_str ("?slha_read_input"), &
          .false., is_known = .true.)
     call var_list_set_log (global%var_list, var_str ("?slha_read_spectrum"), &
```

2877

```
        .false., is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?slha_read_decays"), &
         .false., is_known = .true.)

    call dispatch_slha (global, &
         input = input, spectrum = spectrum, decays = decays)

    write (u, "(A,1x,L1)")  " slha_read_input     =", input
    write (u, "(A,1x,L1)")  " slha_read_spectrum  =", spectrum
    write (u, "(A,1x,L1)")  " slha_read_decays    =", decays

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_14"

  end subroutine dispatch_14
```

22.5 Process Configuration

This module communicates between the toplevel command structure with its
runtime data set and the process-library handling modules which collect the
definition of individual processes. Its primary purpose is to select from the
available matrix-element generating methods and configure the entry in the
process library accordingly.

⟨process_configurations.f90⟩≡
  ⟨*File header*⟩

```
  module process_configurations
```

  ⟨*Use strings*⟩
```
    use diagnostics !NODEP!
    use unit_tests
    use variables
    use models
    use prc_core_def
    use particle_specifiers
    use process_libraries
    use prclib_stacks
    use prc_test
    use prc_omega
    use rt_data
    use dispatch
```

  ⟨*Standard module head*⟩

  ⟨*Process configurations: public*⟩

  ⟨*Process configurations: types*⟩

```
  contains
```

⟨*Process configurations: procedures*⟩

⟨*Process configurations: tests*⟩

```
end module process_configurations
```

22.5.1 Data Type

⟨*Process configurations: public*⟩≡
```
    public :: process_configuration_t
```

⟨*Process configurations: types*⟩≡
```
    type :: process_configuration_t
      type(process_def_entry_t), pointer :: entry => null ()
      type(string_t) :: id
      integer :: num_id = 0
     contains
```
    ⟨*Process configurations: process configuration: TBP*⟩
```
    end type process_configuration_t
```

Initialize a process. We only need the name, the number of incoming particles, and the number of components.

⟨*Process configurations: process configuration: TBP*⟩≡
```
    procedure :: init => process_configuration_init
```

⟨*Process configurations: procedures*⟩≡
```
    subroutine process_configuration_init &
        (config, prc_name, n_in, n_components, global)
      class(process_configuration_t), intent(out) :: config
      type(string_t), intent(in) :: prc_name
      integer, intent(in) :: n_in
      integer, intent(in) :: n_components
      type(rt_data_t), intent(in) :: global
      type(model_t), pointer :: model
      model => global%model
      config%id = prc_name
      allocate (config%entry)
      if (var_list_is_known (global%var_list, var_str ("process_num_id"))) then
         config%num_id = &
              var_list_get_ival (global%var_list, var_str ("process_num_id"))
         call config%entry%init (prc_name, &
              model = model, n_in = n_in, n_components = n_components, &
              num_id = config%num_id)
      else
         call config%entry%init (prc_name, &
              model = model, n_in = n_in, n_components = n_components)
      end if
    end subroutine process_configuration_init
```

Initialize a process component. The details depend on the process method, which determines the type of the process component core. We set the incoming and outgoing particles (as strings, to be interpreted by the process driver). All other information is taken from the variable list.

2879

The dispatcher gets only the names of the particles. The process component definition gets the complete specifiers which contains a polarization flag and names of decay processes, where applicable.

⟨*Process configurations: process configuration: TBP*⟩+≡
```
    procedure :: setup_component => process_configuration_setup_component
```

⟨*Process configurations: procedures*⟩+≡
```
    subroutine process_configuration_setup_component &
        (config, i_component, prt_in, prt_out, global)
      class(process_configuration_t), intent(inout) :: config
      integer, intent(in) :: i_component
      type(prt_spec_t), dimension(:), intent(in) :: prt_in
      type(prt_spec_t), dimension(:), intent(in) :: prt_out
      type(rt_data_t), intent(in) :: global
      type(string_t), dimension(:), allocatable :: prt_str_in
      type(string_t), dimension(:), allocatable :: prt_str_out
      class(prc_core_def_t), allocatable :: core_def
      type(string_t) :: method
      integer :: i
      allocate (prt_str_in  (size (prt_in)))
      allocate (prt_str_out (size (prt_out)))
      forall (i = 1:size (prt_in))  prt_str_in(i)  = prt_in(i)% get_name ()
      forall (i = 1:size (prt_out)) prt_str_out(i) = prt_out(i)%get_name ()
      call dispatch_core_def (core_def, prt_str_in, prt_str_out, global)
      method = var_list_get_sval (global%var_list, var_str ("$method"))
      call config%entry%import_component (i_component, &
           n_out = size (prt_out), &
           prt_in = prt_in, &
           prt_out = prt_out, &
           method = method, &
           variant = core_def)
    end subroutine process_configuration_setup_component
```

Record a process configuration: append it to the currently selected process definition library.

⟨*Process configurations: process configuration: TBP*⟩+≡
```
    procedure :: record => process_configuration_record
```

⟨*Process configurations: procedures*⟩+≡
```
    subroutine process_configuration_record (config, global)
      class(process_configuration_t), intent(inout) :: config
      type(rt_data_t), intent(inout) :: global
      if (associated (global%prclib)) then
         call global%prclib%open ()
         call global%prclib%append (config%entry)
         if (config%num_id /= 0) then
            write (msg_buffer, "(5A,I0,A)") "Process library '", &
                 char (global%prclib%get_name ()), &
                 "': recorded process '", char (config%id), "' (", &
                 config%num_id, ")"
         else
            write (msg_buffer, "(5A)") "Process library '", &
                 char (global%prclib%get_name ()), &
                 "': recorded process '", char (config%id), "'"
```

```
      end if
      call msg_message ()
   else
      call msg_fatal ("Recording process '" // char (config%id) &
            // "': active process library undefined")
   end if
end subroutine process_configuration_record
```

22.5.2 Test

This is the master for calling self-test procedures.

⟨*Process configurations: public*⟩+≡
```
   public :: process_configurations_test
```

⟨*Process configurations: tests*⟩≡
```
   subroutine process_configurations_test (u, results)
     integer, intent(in) :: u
     type(test_results_t), intent(inout) :: results
   ⟨Process configurations: execute tests⟩
 end subroutine process_configurations_test
```

Minimal setup

The workflow for setting up a minimal process configuration with the test matrix element method.

We wrap this in a public procedure, so we can reuse it in later modules. The procedure prepares a process definition list for two processes (one `prc_test` and one `omega` type) and appends this to the process library stack in the global data set.

The `mode` argument determines which processes to build.

The `procname` argument replaces the predefined procname(s).

⟨*Process configurations: public*⟩+≡
```
   public :: prepare_test_library
```

⟨*Process configurations: tests*⟩+≡
```
   subroutine prepare_test_library (global, libname, mode, procname)
     type(rt_data_t), intent(inout), target :: global
     type(string_t), intent(in) :: libname
     integer, intent(in) :: mode
     type(string_t), intent(in), dimension(:), optional :: procname

     type(prclib_entry_t), pointer :: lib
     type(string_t) :: prc_name
     type(string_t), dimension(:), allocatable :: prt_in, prt_out
     integer :: n_components
     type(process_configuration_t) :: prc_config

     allocate (lib)
     call lib%init (libname)
     call global%add_prclib (lib)
```

```
if (btest (mode, 0) .or. btest (mode, 2)) then
   call global%read_model (var_str ("Test"), var_str ("Test.mdl"))
end if

if (btest (mode, 1)) then
   call global%read_model (var_str ("QED"), var_str ("QED.mdl"))
end if

if (btest (mode, 0)) then

   call global%select_model (var_str ("Test"))

   if (present (procname)) then
      prc_name = procname(1)
   else
      prc_name = "prc_config_a"
   end if
   n_components = 1
   allocate (prt_in (2), prt_out (2))
   prt_in = [var_str ("s"), var_str ("s")]
   prt_out = [var_str ("s"), var_str ("s")]

   call var_list_set_string (global%var_list, var_str ("$method"),&
        var_str ("unit_test"), is_known = .true.)

   call prc_config%init (prc_name, size (prt_in), n_components, global)
   call prc_config%setup_component (1, &
        new_prt_spec (prt_in), new_prt_spec (prt_out), global)
   call prc_config%record (global)

   deallocate (prt_in, prt_out)

end if

if (btest (mode, 1)) then

   call global%select_model (var_str ("QED"))

   if (present (procname)) then
      prc_name = procname(2)
   else
      prc_name = "prc_config_b"
   end if
   n_components = 1
   allocate (prt_in (2), prt_out (2))
   prt_in = [var_str ("e+"), var_str ("e-")]
   prt_out = [var_str ("m+"), var_str ("m-")]

   call var_list_set_string (global%var_list, var_str ("$method"),&
        var_str ("omega"), is_known = .true.)

   call prc_config%init (prc_name, size (prt_in), n_components, global)
   call prc_config%setup_component (1, &
        new_prt_spec (prt_in), new_prt_spec (prt_out), global)
```

2882

```
            call prc_config%record (global)

            deallocate (prt_in, prt_out)

         end if

         if (btest (mode, 2)) then

            call global%select_model (var_str ("Test"))

            if (present (procname)) then
               prc_name = procname(1)
            else
               prc_name = "prc_config_a"
            end if
            n_components = 1
            allocate (prt_in (1), prt_out (2))
            prt_in = [var_str ("s")]
            prt_out = [var_str ("f"), var_str ("fbar")]

            call var_list_set_string (global%var_list, var_str ("$method"),&
                 var_str ("unit_test"), is_known = .true.)

            call prc_config%init (prc_name, size (prt_in), n_components, global)
            call prc_config%setup_component (1, &
                 new_prt_spec (prt_in), new_prt_spec (prt_out), global)
            call prc_config%record (global)

            deallocate (prt_in, prt_out)

         end if

    end subroutine prepare_test_library
```

The actual test: the previous procedure with some prelude and postlude. In the global variable list, just before printing we reset the variables where the value may depend on the system and run environment.

⟨*Process configurations: execute tests*⟩≡
```
    call test (process_configurations_1, "process_configurations_1", &
         "test processes", &
         u, results)
```

⟨*Process configurations: tests*⟩+≡
```
    subroutine process_configurations_1 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global

      write (u, "(A)")  "* Test output: process_configurations_1"
      write (u, "(A)")  "*   Purpose: configure test processes"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
```

```
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)

write (u, "(A)")  "* Configure processes as prc_test, model Test"
write (u, "(A)")  "*                       and omega, model QED"
write (u, *)

call var_list_set_int (global%var_list, var_str ("process_num_id"), &
     42, is_known = .true.)
call prepare_test_library (global, var_str ("prc_config_lib_1"), 3)

global%os_data%fc = "Fortran-compiler"
global%os_data%fcflags = "Fortran-flags"

call global%write_libraries (u)

call global%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: process_configurations_1"

end subroutine process_configurations_1
```

O'MEGA options

Slightly extended example where we pass O'MEGA options to the library. The
`prepare_test_library` contents are spelled out.

⟨*Process configurations: execute tests*⟩+≡
```
    call test (process_configurations_2, "process_configurations_2", &
         "omega options", &
         u, results)
```
⟨*Process configurations: tests*⟩+≡
```
    subroutine process_configurations_2 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global

      type(string_t) :: libname
      type(prclib_entry_t), pointer :: lib
      type(string_t) :: prc_name
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      integer :: n_components
      type(process_configuration_t) :: prc_config

      write (u, "(A)")  "* Test output: process_configurations_2"
      write (u, "(A)")  "*   Purpose: configure test processes with options"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
```

2884

```
write (u, "(A)")  "* Configure processes as omega, model QED"
write (u, *)

libname = "prc_config_lib_2"

allocate (lib)
call lib%init (libname)
call global%add_prclib (lib)

call global%read_model (var_str ("QED"), var_str ("QED.mdl"))

prc_name = "prc_config_c"
n_components = 2
allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e+"), var_str ("e-")]
prt_out = [var_str ("m+"), var_str ("m-")]

call var_list_set_string (global%var_list, var_str ("$method"),&
     var_str ("omega"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)

call prc_config%init (prc_name, size (prt_in), n_components, global)

call var_list_set_log (global%var_list, var_str ("?report_progress"), &
     .true., is_known = .true.)
call prc_config%setup_component (1, &
     new_prt_spec (prt_in), new_prt_spec (prt_out), global)

call var_list_set_log (global%var_list, var_str ("?report_progress"), &
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .true., is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$restrictions"),&
     var_str ("3+4~A"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$omega_flags"), &
     var_str ("-fusion:progress_file omega_prc_config.log"), &
     is_known = .true.)
call prc_config%setup_component (2, &
     new_prt_spec (prt_in), new_prt_spec (prt_out), global)

call prc_config%record (global)

deallocate (prt_in, prt_out)

global%os_data%fc = "Fortran-compiler"
global%os_data%fcflags = "Fortran-flags"

call global%write_vars (u, [ &
     var_str ("$model_name"), &
     var_str ("$method"), &
     var_str ("?report_progress"), &
     var_str ("$restrictions"), &
     var_str ("$omega_flags")])
```

2885

```
      write (u, "(A)")
      call global%write_libraries (u)

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_configurations_2"

    end subroutine process_configurations_2
```

22.6 Compilation

This module manages compilation and loading of of process libraries. It is
needed as a separate module because integration depends on it.

⟨compilations.f90⟩≡
  ⟨*File header*⟩

```
  module compilations
```

  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
    use diagnostics !NODEP!
    use limits, only: TAB !NODEP!
    use os_interface
    use unit_tests
    use md5
    use variables
    use models
    use process_libraries
    use prclib_stacks
    use rt_data
    use process_configurations
```

  ⟨*Standard module head*⟩

  ⟨*Compilations: public*⟩

  ⟨*Compilations: types*⟩

```
  contains
```

  ⟨*Compilations: procedures*⟩

  ⟨*Compilations: tests*⟩

```
  end module compilations
```

22.6.1 The data type

The compilation item handles the compilation and loading of a single process library.

⟨*Compilations: types*⟩≡

```
    type :: compilation_item_t
      type(string_t) :: libname
      type(process_library_t), pointer :: lib => null ()
      logical :: recompile_library = .false.
    contains
    ⟨Compilations: compilation item: TBP⟩
    end type compilation_item_t
```

Initialize:

⟨*Compilations: compilation item: TBP*⟩≡

```
    procedure :: init => compilation_item_init
```

⟨*Compilations: procedures*⟩≡

```
    subroutine compilation_item_init (comp, libname, stack, var_list)
      class(compilation_item_t), intent(out) :: comp
      type(string_t), intent(in) :: libname
      type(prclib_stack_t), intent(inout) :: stack
      type(var_list_t), intent(in) :: var_list
      comp%libname = libname
      comp%lib => stack%get_library_ptr (comp%libname)
      if (.not. associated (comp%lib)) then
         call msg_fatal ("Process library '" // char (comp%libname) &
               // "' has not been declared.")
      end if
      comp%recompile_library = &
            var_list_get_lval (var_list, var_str ("?recompile_library"))
    end subroutine compilation_item_init
```

Compile the current library. The `force` flag has the effect that we first delete any previous files, as far as accessible by the current makefile. It also guarantees that previous files not accessible by a makefile will be overwritten.

⟨*Compilations: compilation item: TBP*⟩+≡

```
    procedure :: compile => compilation_item_compile
```

⟨*Compilations: procedures*⟩+≡

```
    subroutine compilation_item_compile (comp, os_data, force, recompile)
      class(compilation_item_t), intent(inout) :: comp
      type(os_data_t), intent(in) :: os_data
      logical, intent(in) :: force, recompile
      if (associated (comp%lib)) then
         call msg_message ("Process library '" &
               // char (comp%libname) // "': compiling ...")
         call comp%lib%configure (os_data)
         if (signal_is_pending ())  return
         call comp%lib%compute_md5sum ()
         call comp%lib%write_makefile (os_data, force)
         if (signal_is_pending ())  return
         if (force) then
            call comp%lib%clean (os_data, distclean = .false.)
```

```
            if (signal_is_pending ())  return
         end if
         call comp%lib%write_driver (force)
         if (signal_is_pending ())  return
         if (recompile) then
            call comp%lib%load (os_data, keep_old_source = .true.)
            if (signal_is_pending ())  return
         end if
         call comp%lib%update_status (os_data)
      end if
    end subroutine compilation_item_compile
```

Load the current library, just after compiling it.

⟨*Compilations: compilation item: TBP*⟩+≡
```
    procedure :: load => compilation_item_load
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_item_load (comp, os_data)
      class(compilation_item_t), intent(inout) :: comp
      type(os_data_t), intent(in) :: os_data
      if (associated (comp%lib)) then
         call comp%lib%load (os_data)
      end if
    end subroutine compilation_item_load
```

Message as a separate call:

⟨*Compilations: compilation item: TBP*⟩+≡
```
    procedure :: success => compilation_item_success
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_item_success (comp)
      class(compilation_item_t), intent(in) :: comp
      if (associated (comp%lib)) then
         call msg_message ("Process library '" // char (comp%libname) &
               // "': ... success.")
      end if
    end subroutine compilation_item_success
```

22.6.2   API for library compilation and loading

This is a shorthand for compiling and loading a single library. The `compilation_item`
object is used only internally.

The `global` data set may actually be local to the caller. The compilation
affects the library specified by its name if it is on the stack, but it does not reset
the currently selected library.

⟨*Compilations: public*⟩≡
```
    public :: compile_library
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compile_library (libname, global)
      type(string_t), intent(in) :: libname
      type(rt_data_t), intent(inout), target :: global
```

```
      type(compilation_item_t) :: comp
      logical :: force, recompile
      force = &
           var_list_get_lval (global%var_list, var_str ("?rebuild_library"))
      recompile = &
           var_list_get_lval (global%var_list, var_str ("?recompile_library"))
      call comp%init (libname, global%prclib_stack, global%var_list)
      call comp%compile (global%os_data, force, recompile)
      if (signal_is_pending ())  return
      call comp%load (global%os_data)
      if (signal_is_pending ())  return
      call comp%success ()
    end subroutine compile_library
```

22.6.3   Compiling static executable

This object handles the creation of a static executable which should contain a
set of static process libraries.

⟨*Compilations: types*⟩+≡
```
    type :: compilation_t
      private
      type(string_t) :: exe_name
      type(string_t), dimension(:), allocatable :: lib_name
     contains
     ⟨Compilations: compilation: TBP⟩
    end type compilation_t
```

Output.

⟨*Compilations: compilation: TBP*⟩≡
```
    procedure :: write => compilation_write
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_write (object, unit)
      class(compilation_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "Compilation object:"
      write (u, "(3x,3A)")  "executable       = '", &
           char (object%exe_name), "'"
      write (u, "(3x,A)", advance="no")  "process libraries ="
      do i = 1, size (object%lib_name)
         write (u, "(1x,3A)", advance="no")  "'", char (object%lib_name(i)), "'"
      end do
      write (u, *)
    end subroutine compilation_write
```

Initialize: we know the names of the executable and of the libraries.

⟨*Compilations: compilation: TBP*⟩+≡
```
    procedure :: init => compilation_init
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_init (compilation, exe_name, lib_name)
  class(compilation_t), intent(out) :: compilation
  type(string_t), intent(in) :: exe_name
  type(string_t), dimension(:), intent(in) :: lib_name
  compilation%exe_name = exe_name
  allocate (compilation%lib_name (size (lib_name)))
  compilation%lib_name = lib_name
end subroutine compilation_init
```

Write the dispatcher subroutine for the compiled libraries. Also write a subrou-
tine which returns the names of the compiled libraries.

⟨*Compilations: compilation: TBP*⟩+≡

```
procedure :: write_dispatcher => compilation_write_dispatcher
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_write_dispatcher (compilation)
  class(compilation_t), intent(in) :: compilation
  type(string_t) :: file
  integer :: u, i
  file = compilation%exe_name // "_prclib_dispatcher.f90"
  call msg_message ("Static executable '" // char (compilation%exe_name) &
       // "': writing library dispatcher")
  u = free_unit ()
  open (u, file = char (file), status="replace", action="write")
  write (u, "(3A)")  "! Whizard: process libraries for executable '", &
       char (compilation%exe_name), "'"
  write (u, "(A)")  "! Automatically generated file, do not edit"
  write (u, "(A)")  "subroutine dispatch_prclib_static " // &
       "(driver, basename, modellibs_ldflags)"
  write (u, "(A)")  "  use iso_varying_string, string_t => varying_string"
  write (u, "(A)")  "  use prclib_interfaces"
  do i = 1, size (compilation%lib_name)
     associate (lib_name => compilation%lib_name(i))
       write (u, "(A)")  "  use " // char (lib_name) // "_driver"
     end associate
  end do
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  class(prclib_driver_t), intent(inout), allocatable &
       &:: driver"
  write (u, "(A)")  "  type(string_t), intent(in) :: basename"
  write (u, "(A)")  "  logical, intent(in), optional :: " // &
       "modellibs_ldflags"
  write (u, "(A)")  "  select case (char (basename))"
  do i = 1, size (compilation%lib_name)
     associate (lib_name => compilation%lib_name(i))
       write (u, "(3A)")  "  case ('", char (lib_name), "')"
       write (u, "(3A)")  "     allocate (", char (lib_name), "_driver_t &
            &:: driver)"
     end associate
  end do
  write (u, "(A)")  "  end select"
  write (u, "(A)")  "end subroutine dispatch_prclib_static"
  write (u, *)
```

```
      write (u, "(A)")  "subroutine get_prclib_static (libname)"
      write (u, "(A)")  "  use iso_varying_string, string_t => varying_string"
      write (u, "(A)")  "  implicit none"
      write (u, "(A)")  "  type(string_t), dimension(:), intent(inout), &
           &allocatable :: libname"
      write (u, "(A,I0,A)")  "  allocate (libname (", &
           size (compilation%lib_name), "))"
      do i = 1, size (compilation%lib_name)
         associate (lib_name => compilation%lib_name(i))
           write (u, "(A,I0,A,A,A)")  "  libname(", i, ") = '", &
                char (lib_name), "'"
         end associate
      end do
      write (u, "(A)")  "end subroutine get_prclib_static"
      close (u)
    end subroutine compilation_write_dispatcher
```

Write the Makefile subroutine for the compiled libraries.

⟨*Compilations: compilation: TBP*⟩+≡
```
    procedure :: write_makefile => compilation_write_makefile
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_write_makefile (compilation, os_data)
      class(compilation_t), intent(in) :: compilation
      type(os_data_t), intent(in) :: os_data
      type(string_t) :: file
      integer :: u, i
      file = compilation%exe_name // ".makefile"
      call msg_message ("Static executable '" // char (compilation%exe_name) &
           // "': writing makefile")
      u = free_unit ()
      open (u, file = char (file), status="replace", action="write")
      write (u, "(3A)")  "# WHIZARD: Makefile for executable '", &
           char (compilation%exe_name), "'"
      write (u, "(A)")  "# Automatically generated file, do not edit"
      write (u, "(A)")  ""
      write (u, "(A)")  "# Executable name"
      write (u, "(A)")  "EXE = " // char (compilation%exe_name)
      write (u, "(A)")  ""
      write (u, "(A)")  "# Compiler"
      write (u, "(A)")  "FC = " // char (os_data%fc)
      write (u, "(A)")  ""
      write (u, "(A)")  "# Included libraries"
      write (u, "(A)")  "FCINCL = " // char (os_data%whizard_includes)
      write (u, "(A)")  ""
      write (u, "(A)")  "# Compiler flags"
      write (u, "(A)")  "FCFLAGS = " // char (os_data%fcflags)
      write (u, "(A)")  "LDFLAGS = " // char (os_data%ldflags)
      write (u, "(A)")  "LDFLAGS_STATIC = " // char (os_data%ldflags_static)
      write (u, "(A)")  "LDFLAGS_HEPMC = " // char (os_data%ldflags_hepmc)
      write (u, "(A)")  "LDFLAGS_HOPPET = " // char (os_data%ldflags_hoppet)
      write (u, "(A)")  "LDWHIZARD = " // char (os_data%whizard_ldflags)
      write (u, "(A)")  ""
      write (u, "(A)")  "# Libtool"
```

```
       write (u, "(A)") "LIBTOOL = " // char (os_data%whizard_libtool)
       write (u, "(A)") "FCOMPILE = $(LIBTOOL) --tag=FC --mode=compile"
       write (u, "(A)") "LINK = $(LIBTOOL) --tag=FC --mode=link"
       write (u, "(A)") ""
       write (u, "(A)") "# Compile commands (default)"
       write (u, "(A)") "LTFCOMPILE = $(FCOMPILE) $(FC) -c $(FCINCL) $(FCFLAGS)"
       write (u, "(A)") ""
       write (u, "(A)") "# Default target"
       write (u, "(A)") "all: link"
       write (u, "(A)") ""
       write (u, "(A)") "# Libraries"
       do i = 1, size (compilation%lib_name)
          associate (lib_name => compilation%lib_name(i))
            write (u, "(A)") "LIBRARIES += " // char (lib_name) // ".la"
            write (u, "(A)") char (lib_name) // ".la:"
            write (u, "(A)") TAB // "$(MAKE) -f " // char (lib_name) // ".makefile"
          end associate
       end do
       write (u, "(A)") ""
       write (u, "(A)") "# Library dispatcher"
       write (u, "(A)") "DISP = $(EXE)_prclib_dispatcher"
       write (u, "(A)") "$(DISP).lo: $(DISP).f90 $(LIBRARIES)"
       write (u, "(A)") TAB // "$(LTFCOMPILE) $<"
       write (u, "(A)") ""
       write (u, "(A)") "# Executable"
       write (u, "(A)") "$(EXE): $(DISP).lo $(LIBRARIES)"
       write (u, "(A)") TAB // "$(LINK) $(FC) -static-libtool-libs $(FCFLAGS) \"
       write (u, "(A)") TAB // "   $(LDWHIZARD) $(LDFLAGS) $(LDFLAGS_STATIC) \"
       write (u, "(A)") TAB // "     -o $(EXE) $^ $(LDFLAGS_HEPMC) $(LDFLAGS_HOPPET)"
       write (u, "(A)") ""
       write (u, "(A)") "# Main targets"
       write (u, "(A)") "link: compile $(EXE)"
       write (u, "(A)") "compile: $(LIBRARIES) $(DISP).lo"
       write (u, "(A)") ".PHONY: link compile"
       write (u, "(A)") ""
       write (u, "(A)") "# Cleanup targets"
       write (u, "(A)") "clean-exe:"
       write (u, "(A)") TAB // "rm -f $(EXE)"
       write (u, "(A)") "clean-objects:"
       write (u, "(A)") TAB // "rm -f $(DISP).lo"
       write (u, "(A)") "clean-source:"
       write (u, "(A)") TAB // "rm -f $(DISP).f90"
       write (u, "(A)") "clean-makefile:"
       write (u, "(A)") TAB // "rm -f $(EXE).makefile"
       write (u, "(A)") ""
       write (u, "(A)") "clean: clean-exe clean-objects clean-source"
       write (u, "(A)") "distclean: clean clean-makefile"
       write (u, "(A)") ".PHONY: clean distclean"
       close (u)
  end subroutine compilation_write_makefile
```

Compile the dispatcher source code.

⟨*Compilations: compilation: TBP*⟩+≡

```
    procedure :: make_compile => compilation_make_compile
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_make_compile (compilation, os_data)
      class(compilation_t), intent(in) :: compilation
      type(os_data_t), intent(in) :: os_data
      call os_system_call ("make compile " // os_data%makeflags &
          // " -f " // compilation%exe_name // ".makefile")
    end subroutine compilation_make_compile
```

Link the dispatcher together with all matrix-element code and the WHIZARD and
O'MEGA main libraries, to generate a static executable.

⟨*Compilations: compilation: TBP*⟩+≡
```
    procedure :: make_link => compilation_make_link
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_make_link (compilation, os_data)
      class(compilation_t), intent(in) :: compilation
      type(os_data_t), intent(in) :: os_data
      call os_system_call ("make link " // os_data%makeflags &
          // " -f " // compilation%exe_name // ".makefile")
    end subroutine compilation_make_link
```

Cleanup.

⟨*Compilations: compilation: TBP*⟩+≡
```
    procedure :: make_clean_exe => compilation_make_clean_exe
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compilation_make_clean_exe (compilation, os_data)
      class(compilation_t), intent(in) :: compilation
      type(os_data_t), intent(in) :: os_data
      call os_system_call ("make clean-exe " // os_data%makeflags &
          // " -f " // compilation%exe_name // ".makefile")
    end subroutine compilation_make_clean_exe
```

22.6.4   API for executable compilation

This is a shorthand for compiling and loading an executable, including the
enclosed libraries. The `compilation` object is used only internally.

The `global` data set may actually be local to the caller. The compilation
affects the library specified by its name if it is on the stack, but it does not reset
the currently selected library.

⟨*Compilations: public*⟩+≡
```
    public :: compile_executable
```

⟨*Compilations: procedures*⟩+≡
```
    subroutine compile_executable (exename, libname, global)
      type(string_t), intent(in) :: exename
      type(string_t), dimension(:), intent(in) :: libname
      type(rt_data_t), intent(inout), target :: global
      type(compilation_t) :: compilation
      type(compilation_item_t) :: item
```

```
logical :: force, recompile
integer :: i
force = &
     var_list_get_lval (global%var_list, var_str ("?rebuild_library"))
recompile = &
     var_list_get_lval (global%var_list, var_str ("?recompile_library"))
call compilation%init (exename, [libname])
if (signal_is_pending ())  return
call compilation%write_dispatcher ()
if (signal_is_pending ())  return
call compilation%write_makefile (global%os_data)
if (signal_is_pending ())  return
do i = 1, size (libname)
   call item%init (libname(i), global%prclib_stack, global%var_list)
   call item%compile (global%os_data, force=force, recompile=recompile)
   if (signal_is_pending ())  return
   call item%success ()
end do
call compilation%make_compile (global%os_data)
if (signal_is_pending ())  return
call compilation%make_link (global%os_data)
end subroutine compile_executable
```

22.6.5   Test

This is the master for calling self-test procedures.

⟨*Compilations: public*⟩+≡
```
public :: compilations_test
```

⟨*Compilations: tests*⟩≡
```
subroutine compilations_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Compilations: execute tests⟩
end subroutine compilations_test
```

Intrinsic Matrix Element

Compile an intrinsic test matrix element (`prc_test` type).

   Note: In this and the following test, we reset the Fortran compiler and flag variables immediately before they are printed, so the test is portable.

⟨*Compilations: execute tests*⟩≡
```
call test (compilations_1, "compilations_1", &
     "intrinsic test processes", &
     u, results)
```

⟨*Compilations: tests*⟩+≡
```
subroutine compilations_1 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global
```

```
      write (u, "(A)")   "* Test output: compilations_1"
      write (u, "(A)")   "*   Purpose: configure and compile test process"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()

      libname = "compilation_1"
      procname = "prc_comp_1"
      call prepare_test_library (global, libname, 1, [procname])

      call compile_library (libname, global)

      call global%write_libraries (u)

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: compilations_1"

    end subroutine compilations_1
```

External Matrix Element

Compile an external test matrix element (`omega` type)

⟨*Compilations: execute tests*⟩+≡
```
    call test (compilations_2, "compilations_2", &
        "external process (omega)", &
        u, results)
```

⟨*Compilations: tests*⟩+≡
```
    subroutine compilations_2 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global

      write (u, "(A)")   "* Test output: compilations_2"
      write (u, "(A)")   "*   Purpose: configure and compile test process"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)

      libname = "compilation_2"
      procname = "prc_comp_2"
      call prepare_test_library (global, libname, 2, [procname,procname])
```

```
      call compile_library (libname, global)

      call global%write_libraries (u, libpath = .false.)

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: compilations_2"

    end subroutine compilations_2
```

External Matrix Element

Compile an external test matrix element (`omega` type) and create driver files for a static executable.

⟨*Compilations: execute tests*⟩+≡
```
    call test (compilations_3, "compilations_3", &
        "static executable: driver", &
        u, results)
```

⟨*Compilations: tests*⟩+≡
```
    subroutine compilations_3 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname, exename
      type(rt_data_t), target :: global
      type(compilation_t) :: compilation
      integer :: u_file
      character(80) :: buffer

      write (u, "(A)")  "* Test output: compilations_3"
      write (u, "(A)")  "*   Purpose: make static executable"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize library"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)

      libname = "compilations_3_lib"
      procname = "prc_comp_3"
      exename = "compilations_3"

      call prepare_test_library (global, libname, 2, [procname,procname])

      call compilation%init (exename, [libname])
      call compilation%write (u)

      write (u, "(A)")
```

```
      write (u, "(A)")  "* Write dispatcher"
      write (u, "(A)")

      call compilation%write_dispatcher ()

      u_file = free_unit ()
      open (u_file, file = char (exename) // "_prclib_dispatcher.f90", &
            status = "old", action = "read")
      do
         read (u_file, "(A)", end = 1)  buffer
         write (u, "(A)")  trim (buffer)
      end do
1     close (u_file)

      write (u, "(A)")
      write (u, "(A)")  "* Write Makefile"
      write (u, "(A)")

      associate (os_data => global%os_data)
        os_data%fc = "fortran-compiler"
        os_data%whizard_includes = "my-includes"
        os_data%fcflags = "my-fcflags"
        os_data%ldflags = "my-ldflags"
        os_data%ldflags_static = "my-ldflags-static"
        os_data%ldflags_hepmc = "my-ldflags-hepmc"
        os_data%ldflags_hoppet = "my-ldflags-hoppet"
        os_data%whizard_ldflags = "my-ldwhizard"
        os_data%whizard_libtool = "my-libtool"
      end associate

      call compilation%write_makefile (global%os_data)

      open (u_file, file = char (exename) // ".makefile", &
            status = "old", action = "read")
      do
         read (u_file, "(A)", end = 2)  buffer
         write (u, "(A)")  trim (buffer)
      end do
2     close (u_file)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: compilations_3"

    end subroutine compilations_3
```

22.6.6 Test static build

The tests for building a static executable are separate, since they should be skipped if the WHIZARD build itself has static libraries disabled.

⟨*Compilations: public*⟩+≡
```
    public :: compilations_static_test
```

⟨*Compilations: tests*⟩+≡
```
    subroutine compilations_static_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Compilations: static tests⟩
  end subroutine compilations_static_test
```

External Matrix Element

Compile an external test matrix element (omega type) and incorporate this in a new static WHIZARD executable.

⟨*Compilations: static tests*⟩≡
```
    call test (compilations_static_1, "compilations_static_1", &
        "static executable: compilation", &
        u, results)
```

⟨*Compilations: tests*⟩+≡
```
    subroutine compilations_static_1 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname, exename
      type(rt_data_t), target :: global
      type(compilation_item_t) :: item
      type(compilation_t) :: compilation
      logical :: exist

      write (u, "(A)")  "* Test output: compilations_static_1"
      write (u, "(A)")  "*   Purpose: make static executable"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize library"

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)

      libname = "compilations_static_1_lib"
      procname = "prc_comp_stat_1"
      exename = "compilations_static_1"

      call prepare_test_library (global, libname, 2, [procname,procname])

      call compilation%init (exename, [libname])

      write (u, "(A)")
```

2898

```
    write (u, "(A)")  "* Write dispatcher"

    call compilation%write_dispatcher ()

    write (u, "(A)")
    write (u, "(A)")  "* Write Makefile"

    call compilation%write_makefile (global%os_data)

    write (u, "(A)")
    write (u, "(A)")  "* Build libraries"

    call item%init (libname, global%prclib_stack, global%var_list)
    call item%compile (global%os_data, force=.true., recompile=.false.)
    call item%success ()

    write (u, "(A)")
    write (u, "(A)")  "* Check executable (should be absent)"
    write (u, "(A)")

    call compilation%make_clean_exe (global%os_data)
    inquire (file = char (exename), exist = exist)
    write (u, "(A,A,L1)")  char (exename), " exists = ", exist

    write (u, "(A)")
    write (u, "(A)")  "* Build executable"
    write (u, "(A)")

    call compilation%make_compile (global%os_data)
    call compilation%make_link (global%os_data)

    write (u, "(A)")  "* Check executable (should be present)"
    write (u, "(A)")

    inquire (file = char (exename), exist = exist)
    write (u, "(A,A,L1)")  char (exename), " exists = ", exist

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call compilation%make_clean_exe (global%os_data)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_static_1"

  end subroutine compilations_static_1
```

External Matrix Element

Compile an external test matrix element (`omega` type) and incorporate this in a new static WHIZARD executable. In this version, we use the wrapper `compile_executable` procedure.

⟨*Compilations: static tests*⟩+≡
```
    call test (compilations_static_2, "compilations_static_2", &
         "static executable: shortcut", &
         u, results)
```

⟨*Compilations: tests*⟩+≡
```
    subroutine compilations_static_2 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname, exename
      type(rt_data_t), target :: global
      type(compilation_item_t) :: item
      type(compilation_t) :: compilation
      logical :: exist
      integer :: u_file

      write (u, "(A)")  "* Test output: compilations_static_2"
      write (u, "(A)")  "*   Purpose: make static executable"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize library and compile"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
           .false., is_known = .true.)

      libname = "compilations_static_2_lib"
      procname = "prc_comp_stat_2"
      exename = "compilations_static_2"

      call prepare_test_library (global, libname, 2, [procname,procname])

      call compile_executable (exename, [libname], global)

      write (u, "(A)")  "* Check executable (should be present)"
      write (u, "(A)")

      inquire (file = char (exename), exist = exist)
      write (u, "(A,A,L1)")  char (exename), " exists = ", exist

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      u_file = free_unit ()
      open (u_file, file = char (exename), status = "old", action = "write")
      close (u_file, status = "delete")

      call global%final ()
```

```
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_static_2"

  end subroutine compilations_static_2
```

22.7 Integration

This module manages phase space setup, matrix-element evaluation and integration, as far as it is not done by lower-level routines, in particular in the `processes` module.

⟨integrations.f90⟩≡
  ⟨*File header*⟩

```
  module integrations
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
```
  use diagnostics !NODEP!
  use unit_tests
  use os_interface
  use cputime
  use sm_qcd
  use ifiles
  use lexers
  use parser
  use flavors
  use pdg_arrays
  use variables
  use expressions
  use models
  use interactions
  use sf_mappings
  use sf_base
  use phs_base
  use mappings
  use phs_forests
  use phs_wood
  use rng_base
  use mci_base
  use process_libraries
  use prc_core
  use processes
  use process_stacks
  use iterations
  use rt_data
  use dispatch
  use process_configurations
  use compilations
```

⟨*Standard module head*⟩

⟨*Integrations: public*⟩

⟨*Integrations: types*⟩

```
contains
```

⟨*Integrations: procedures*⟩

⟨*Integrations: tests*⟩

```
end module integrations
```

22.7.1   The integration type

This type holds all relevant data, the integration methods operates on this. In contrast to the `simulation_t` introduced later, the `integration_t` applies to a single process.

⟨*Integrations: public*⟩≡
```
    public :: integration_t
```

⟨*Integrations: types*⟩≡
```
    type :: integration_t
      private
      type(string_t) :: process_id
      type(string_t) :: run_id
      type(process_t), pointer :: process => null ()
      logical :: rebuild_phs = .false.
      logical :: ignore_phs_mismatch = .false.
      logical :: phs_only = .false.
      logical :: process_has_me = .true.
      integer :: n_calls_test = 0
      logical :: vis_history = .true.
      type(string_t) :: history_filename
      type(string_t) :: log_filename
     contains
     ⟨Integrations: integration: TBP⟩
    end type integration_t
```

22.7.2   Initialization

Initialization, first part: Create and initialize a process entry and push it on the stack. Only the basic process data (ID, run ID, etc.) are configured, we postpone the rest.

⟨*Integrations: integration: TBP*⟩≡
```
    procedure :: create_process => integration_create_process
```

⟨*Integrations: procedures*⟩≡
```
    subroutine integration_create_process (intg, process_id, global) !, verbose)
      class(integration_t), intent(out) :: intg
```

```
      type(rt_data_t), intent(inout), target :: global
      type(string_t), intent(in) :: process_id
      type(var_list_t), pointer :: var_list
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      type(process_entry_t), pointer :: process_entry

      var_list => global%var_list
      intg%process_id = process_id
      intg%run_id = var_list_get_sval (var_list, var_str ("$run_id"))

      call dispatch_qcd (qcd, global)
      call dispatch_rng_factory (rng_factory, global)

      allocate (process_entry)
      call process_entry%init (intg%process_id, intg%run_id, global%prclib, &
           global%os_data, qcd, rng_factory, global%model_list)
      call global%process_stack%push (process_entry)

    end subroutine integration_create_process
```

Complete process configuration.

   If the `process` argument is given, use it. Otherwise, take the process object from the stack.

⟨*Integrations: integration: TBP*⟩+≡
```
    procedure :: setup_process => integration_setup_process
```

⟨*Integrations: procedures*⟩+≡
```
    subroutine integration_setup_process (intg, global, process, verbose)
      class(integration_t), intent(inout) :: intg
      type(rt_data_t), intent(inout), target :: global
      type(process_t), intent(in), target, optional :: process
      logical, intent(in), optional :: verbose

      class(prc_core_t), allocatable :: core_template
      class(phs_config_t), allocatable :: phs_config_template
      type(phs_parameters_t) :: phs_par
      type(mapping_defaults_t) :: mapping_defs
      class(mci_t), allocatable :: mci_template
      type(qcd_t) :: qcd
      integer :: n_components, n_in, i_component
      type(pdg_array_t), dimension(:,:), allocatable :: pdg_prc
      type(process_component_def_t), pointer :: config
      type(helicity_selection_t), allocatable :: helicity_selection
      real(default) :: sqrts
      logical :: decay_rest_frame, use_color_factors
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_prop_t) :: sf_prop
      type(sf_channel_t), dimension(:), allocatable :: sf_channel
      type(phs_channel_collection_t) :: phs_channel_collection
      logical :: sf_trace
      type(string_t) :: sf_string, sf_trace_file
      logical :: verb
```

2903

```
verb = .true.; if (present (verbose))  verb = verbose

if (present (process)) then
   intg%process => process
   intg%process_id = process%get_id ()
   intg%run_id = process%get_run_id ()
else
   intg%process => global%process_stack%get_process_ptr (intg%process_id)
end if

call intg%process%set_var_list (global%var_list)

intg%rebuild_phs = &
     var_list_get_lval (global%var_list, var_str ("?rebuild_phase_space"))
intg%ignore_phs_mismatch = &
     .not. var_list_get_lval (global%var_list, var_str ("?check_phs_file"))
intg%phs_only = var_list_get_lval &
     (global%var_list, var_str ("?phs_only"))
phs_par%m_threshold_s = var_list_get_rval &
     (global%var_list, var_str ("phs_threshold_s"))
phs_par%m_threshold_t = var_list_get_rval &
     (global%var_list, var_str ("phs_threshold_t"))
phs_par%off_shell = var_list_get_ival &
     (global%var_list, var_str ("phs_off_shell"))
phs_par%keep_nonresonant = var_list_get_lval &
     (global%var_list, var_str ("?phs_keep_nonresonant"))
phs_par%t_channel = var_list_get_ival &
     (global%var_list, var_str ("phs_t_channel"))
mapping_defs%energy_scale = var_list_get_rval &
     (global%var_list, var_str ("phs_e_scale"))
mapping_defs%invariant_mass_scale = var_list_get_rval &
     (global%var_list, var_str ("phs_m_scale"))
mapping_defs%momentum_transfer_scale = var_list_get_rval &
     (global%var_list, var_str ("phs_q_scale"))
mapping_defs%step_mapping = var_list_get_lval &
     (global%var_list, var_str ("?phs_step_mapping"))
mapping_defs%step_mapping_exp = var_list_get_lval &
     (global%var_list, var_str ("?phs_step_mapping_exp"))
mapping_defs%enable_s_mapping = var_list_get_lval &
     (global%var_list, var_str ("?phs_s_mapping"))

call dispatch_phs (phs_config_template, global, &
     intg%process_id, mapping_defs, phs_par)

intg%n_calls_test = &
     var_list_get_ival (global%var_list, var_str ("n_calls_test"))

!!! We avoid two dots in the filename due to a bug in certain MetaPost versions.
if (intg%run_id /= "") then
   intg%history_filename = intg%process_id // "." // intg%run_id &
        // "-history"
   intg%log_filename = intg%process_id // "." // intg%run_id // ".log"
else
   intg%history_filename = intg%process_id // "-history"
```

```
   intg%log_filename = intg%process_id // ".log"
end if

call dispatch_mci (mci_template, global, intg%process_id)

if (verb) then
   call msg_message ("Initializing integration for process " &
        // char (intg%process_id) // ":")
   if (intg%run_id /= "") then
      call msg_message ("Run ID = " // '"' // char (intg%run_id) // '"')
   end if
end if

helicity_selection = global%get_helicity_selection ()

intg%vis_history = &
     var_list_get_lval (global%var_list, var_str ("?vis_history"))
use_color_factors = var_list_get_lval &
     (global%var_list, var_str ("?read_color_factors"))

call dispatch_qcd (qcd, global)

n_components = intg%process%get_n_components ()
n_in = intg%process%get_n_in ()

do i_component = 1, n_components
   config => intg%process%get_component_def_ptr (i_component)
   call dispatch_core (core_template, config%get_core_def_ptr (), &
        intg%process%get_model_ptr (), helicity_selection, qcd, &
        use_color_factors)
   call intg%process%init_component &
        (i_component, core_template, mci_template, phs_config_template)
   deallocate (core_template)
end do

call intg%process%write (screen = .true.)

intg%process_has_me = intg%process%has_matrix_element ()
if (.not. intg%process_has_me) then
   call msg_warning ("Process '" &
        // char (intg%process_id) // "': matrix element vanishes")
end if

sqrts = global%get_sqrts ()
decay_rest_frame = &
     var_list_get_lval (global%var_list, var_str ("?decay_rest_frame"))
if (intg%process_has_me) then
   if (global%beam_structure%is_set ()) then
      call intg%process%setup_beams_beam_structure &
           (global%beam_structure, sqrts, global%model, decay_rest_frame)
    else if (n_in == 2) then
      call intg%process%setup_beams_sqrts &
           (sqrts, global%beam_structure)
   else
```

```
         call intg%process%setup_beams_decay &
              (decay_rest_frame, global%beam_structure)
      end if
   end if
   call intg%process%check_masses ()
   if (intg%process_has_me)  call intg%process%beams_startup_message &
        (beam_structure = global%beam_structure)

   if (intg%process_has_me) then
      call intg%process%get_pdg_in (pdg_prc)
   else
      allocate (pdg_prc (n_in, n_components))
      pdg_prc = 0
   end if
   call dispatch_sf_config (sf_config, sf_prop, global, pdg_prc)
   sf_trace = &
        var_list_get_lval (global%var_list, var_str ("?sf_trace"))
   sf_trace_file = &
        var_list_get_sval (global%var_list, var_str ("$sf_trace_file"))
   if (sf_trace) then
      call intg%process%init_sf_chain (sf_config, sf_trace_file)
   else
      call intg%process%init_sf_chain (sf_config)
   end if

   if (intg%process_has_me) then
      call intg%process%configure_phs (intg%rebuild_phs, intg%ignore_phs_mismatch)
      if (size (sf_config) > 0) then
         call intg%process%collect_channels (phs_channel_collection)
      else if (intg%process%contains_trivial_component ()) then
         call msg_fatal ("Integrate: 2 -> 1 process can't be handled &
              &with fixed-energy beams")
      end if

      call dispatch_sf_channels &
           (sf_channel, sf_string, sf_prop, phs_channel_collection, global)
      if (allocated (sf_channel)) then
         if (size (sf_channel) > 0) then
            call intg%process%set_sf_channel (sf_channel)
         end if
      end if
      call phs_channel_collection%final ()
      call intg%process%sf_startup_message (sf_string)
   end if

   call intg%process%setup_mci ()
   call intg%process%setup_terms ()

   if (associated (global%pn%cuts_lexpr)) then
      if (verb)  call msg_message ("Applying user-defined cuts.")
      call intg%process%set_cuts (global%pn%cuts_lexpr)
   else
      if (verb)  call msg_warning ("No cuts have been defined.")
   end if
```

```
if (associated (global%pn%scale_expr) .and. verb) then
   call msg_message ("Using user-defined general scale.")
   call intg%process%set_scale (global%pn%scale_expr)
end if
if (associated (global%pn%fac_scale_expr) .and. verb) then
   call msg_message ("Using user-defined factorization scale.")
   call intg%process%set_fac_scale (global%pn%fac_scale_expr)
end if
if (associated (global%pn%ren_scale_expr) .and. verb) then
   call msg_message ("Using user-defined renormalization scale.")
   call intg%process%set_ren_scale (global%pn%ren_scale_expr)
end if
if (associated (global%pn%weight_expr) .and. verb) then
   call msg_message ("Using user-defined reweighting factor.")
   call intg%process%set_weight (global%pn%weight_expr)
end if

call intg%process%compute_md5sum ()

end subroutine integration_setup_process
```

22.7.3 Integration

Integrate: do the final integration. Here, we do a multi-iteration integration. Again, we skip iterations that are already on file. Record the results in the global variable list.

⟨*Integrations: integration: TBP*⟩+≡
```
procedure :: evaluate => integration_evaluate
```

⟨*Integrations: procedures*⟩+≡
```
subroutine integration_evaluate &
     (intg, process_instance, i_mci, pass, it_list, pacify)
  class(integration_t), intent(inout) :: intg
  type(process_instance_t), intent(inout), target :: process_instance
  integer, intent(in) :: i_mci
  integer, intent(in) :: pass
  type(iterations_list_t), intent(in) :: it_list
  logical, intent(in), optional :: pacify
  integer :: n_calls, n_it
  logical :: adapt_grids, adapt_weights, final

  n_it = it_list%get_n_it (pass)
  n_calls = it_list%get_n_calls (pass)
  adapt_grids = it_list%adapt_grids (pass)
  adapt_weights = it_list%adapt_weights (pass)
  final = pass == it_list%get_n_pass ()

  call intg%process%integrate (process_instance, &
       i_mci, n_it, n_calls, adapt_grids, adapt_weights, &
       final, pacify)

end subroutine integration_evaluate
```

In case the user has not provided a list of iterations, make a reasonable default. This can depend on the process. The usual approach is to define two distinct passes, one for adaptation and one for integration.

⟨*Integrations: integration: TBP*⟩+≡
```
    procedure :: make_iterations_list => integration_make_iterations_list
```

⟨*Integrations: procedures*⟩+≡
```
    subroutine integration_make_iterations_list (intg, it_list)
      class(integration_t), intent(in) :: intg
      type(iterations_list_t), intent(out) :: it_list
      integer :: pass, n_pass
      integer, dimension(:), allocatable :: n_it, n_calls
      logical, dimension(:), allocatable :: adapt_grids, adapt_weights
      n_pass = intg%process%get_n_pass_default ()
      allocate (n_it (n_pass), n_calls (n_pass))
      allocate (adapt_grids (n_pass), adapt_weights (n_pass))
      do pass = 1, n_pass
         n_it(pass)           = intg%process%get_n_it_default (pass)
         n_calls(pass)        = intg%process%get_n_calls_default (pass)
         adapt_grids(pass)    = intg%process%adapt_grids_default (pass)
         adapt_weights(pass) = intg%process%adapt_weights_default (pass)
      end do
      call it_list%init (n_it, n_calls, &
           adapt_grids = adapt_grids, adapt_weights = adapt_weights)
    end subroutine integration_make_iterations_list
```

22.7.4 API for integration objects

This initializer does everything except assigning cuts/scale/weight expressions. If ok is false, initialization failed, and the integration should be skipped.

If me_only is set, prepare just matrix-element evaluation: no phase space, no iterations.

⟨*Integrations: integration: TBP*⟩+≡
```
    procedure :: init => integration_init
```

⟨*Integrations: procedures*⟩+≡
```
    subroutine integration_init (intg, process_id, global)
      class(integration_t), intent(out) :: intg
      type(string_t), intent(in) :: process_id
      type(rt_data_t), intent(inout), target :: global

      call intg%create_process (process_id, global)
      call intg%setup_process (global)
    end subroutine integration_init
```

Do the integration for a single process, both warmup and final evaluation. The eff_reset flag is to suppress numerical noise in the graphical output of the integration history.

⟨*Integrations: integration: TBP*⟩+≡
```
    procedure :: integrate => integration_integrate
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_integrate (intg, global, eff_reset)
  class(integration_t), intent(inout) :: intg
  type(rt_data_t), intent(inout), target :: global
  logical, intent(in), optional :: eff_reset
  type(string_t) :: log_filename
  type(process_instance_t), allocatable, target :: process_instance
  type(iterations_list_t) :: it_list
  logical :: pacify
  integer :: pass, i_mci, n_mci, n_pass

  allocate (process_instance)
  call process_instance%init (intg%process)

  call openmp_set_num_threads_verbose &
       (var_list_get_ival (global%var_list, "openmp_num_threads"), &
        var_list_get_lval (global%var_list, "?openmp_logging"))
  pacify = var_list_get_lval (global%var_list, var_str ("?pacify"))

  n_mci = intg%process%get_n_mci ()
  if (n_mci == 1) then
     write (msg_buffer, "(A,A,A)") &
          "Starting integration for process '", &
          char (intg%process%get_id ()), "'"
     call msg_message ()
  end if
  do i_mci = 1, n_mci
     if (n_mci > 1) then
        write (msg_buffer, "(A,A,A,I0)") &
             "Starting integration for process '", &
             char (intg%process%get_id ()), "' part ", i_mci
        call msg_message ()
     end if
     n_pass = global%it_list%get_n_pass ()
     if (n_pass == 0) then
        call msg_message ("Integrate: iterations not specified, &
             &using default")
        call intg%make_iterations_list (it_list)
        n_pass = it_list%get_n_pass ()
     else
        it_list = global%it_list
     end if
     call msg_message ("Integrate: " // char (it_list%to_string ()))
     do pass = 1, n_pass
        call intg%evaluate (process_instance, i_mci, pass, it_list, pacify)
        if (signal_is_pending ())  return
     end do
     call intg%process%final_integration (i_mci)
     if (intg%vis_history) then
        call intg%process%display_integration_history &
             (i_mci, intg%history_filename, global%os_data, eff_reset)
     end if
     if (global%logfile == intg%log_filename) then
        if (intg%run_id /= "") then
```

```
              log_filename = intg%process_id // "." // intg%run_id // &
                    ".var.log"
          else
              log_filename = intg%process_id // ".var.log"
          end if
          call msg_message ("Name clash for global logfile and process log: ", &
              arr =[var_str ("| Renaming log file from ") // global%logfile, &
                    var_str ("|   to ") // log_filename // var_str (" .")])
      else
          log_filename = intg%log_filename
      end if
      call intg%process%write_logfile (i_mci, log_filename)
    end do

    if (n_mci > 1) then
        call msg_message ("Integrate: sum of all components")
        call intg%process%display_summed_results ()
    end if

    call process_instance%final ()
    deallocate (process_instance)

  end subroutine integration_integrate
```

Do a dummy integration for a process which could not be initialized (e.g., has no matrix element). The result is zero.

⟨*Integrations: integration: TBP*⟩+≡
```
    procedure :: integrate_dummy => integration_integrate_dummy
```

⟨*Integrations: procedures*⟩+≡
```
    subroutine integration_integrate_dummy (intg)
      class(integration_t), intent(inout) :: intg
      call intg%process%integrate_dummy ()
    end subroutine integration_integrate_dummy
```

Just sample the matrix element under realistic conditions (but no cuts); throw away the results.

⟨*Integrations: integration: TBP*⟩+≡
```
    procedure :: sampler_test => integration_sampler_test
```

⟨*Integrations: procedures*⟩+≡
```
    subroutine integration_sampler_test (intg, global)
      class(integration_t), intent(inout) :: intg
      type(rt_data_t), intent(inout), target :: global
      type(process_instance_t), allocatable, target :: process_instance
      integer :: n_mci, i_mci
      type(timer_t) :: timer_mci, timer_tot
      real(default) :: t_mci, t_tot
      allocate (process_instance)
      call process_instance%init (intg%process)
      n_mci = intg%process%get_n_mci ()
      if (n_mci == 1) then
          write (msg_buffer, "(A,A,A)") &
                "Test: probing process '", &
```

```
            char (intg%process%get_id ()), "'"
        call msg_message ()
     end if
     call timer_tot%start ()
     do i_mci = 1, n_mci
        if (n_mci > 1) then
           write (msg_buffer, "(A,A,A,I0)") &
                "Test: probing process '", &
                char (intg%process%get_id ()), "' part ", i_mci
           call msg_message ()
        end if
        call timer_mci%start ()
        call intg%process%sampler_test &
             (process_instance, i_mci, intg%n_calls_test)
        call timer_mci%stop ()
        t_mci = timer_mci
        write (msg_buffer, "(A,ES12.5)")  "Test: " &
             // "time in seconds (wallclock): ", t_mci
        call msg_message ()
     end do
     call timer_tot%stop ()
     t_tot = timer_tot
     if (n_mci > 1) then
        write (msg_buffer, "(A,ES12.5)")  "Test: " &
             // "total time      (wallclock): ", t_tot
        call msg_message ()
     end if
     call process_instance%final ()
   end subroutine integration_sampler_test
```

Simply integrate, do a dummy integration if necessary. The `integration` object exists only internally.

The `init_only` flag can be used to skip the actual integration part. We will end up with a process object that is completely initialized, including phase space configuration.

The `eff_reset` flag is to suppress numerical noise in the visualization of the integration history.

⟨*Integrations: public*⟩+≡
```
    public :: integrate_process
```

⟨*Integrations: procedures*⟩+≡
```
    subroutine integrate_process (process_id, global, init_only, eff_reset)
      type(string_t), intent(in) :: process_id
      type(rt_data_t), intent(inout), target :: global
      logical, intent(in), optional :: init_only, eff_reset
      type(string_t) :: prclib_name
      type(integration_t) :: intg
      character(32) :: buffer

      if (.not. associated (global%prclib)) then
         call msg_fatal ("Integrate: current process library is undefined")
         return
      end if
```

```
      if (.not. global%prclib%is_active ()) then
         call msg_message ("Integrate: current process library needs compilation")
         prclib_name = global%prclib%get_name ()
         call compile_library (prclib_name, global)
         if (signal_is_pending ())  return
         call msg_message ("Integrate: compilation done")
      end if

      call intg%init (process_id, global)
      if (signal_is_pending ())  return

      if (present (init_only)) then
         if (init_only) return
      end if

      if (intg%n_calls_test > 0) then
         write (buffer, "(I0)")  intg%n_calls_test
         call msg_message ("Integrate: test (" // trim (buffer) // " calls) ...")
         call intg%sampler_test (global)
         call msg_message ("Integrate: ... test complete.")
         if (signal_is_pending ())  return
      end if

      if (intg%phs_only) then
         call msg_message ("Integrate: phase space only, skipping integration")
      else
         if (intg%process_has_me) then
            call intg%integrate (global, eff_reset)
         else
            call intg%integrate_dummy ()
         end if
      end if
   end subroutine integrate_process
```

22.7.5 Test

This is the master for calling self-test procedures.

⟨*Integrations: public*⟩+≡
```
    public :: integrations_test
```

⟨*Integrations: tests*⟩≡
```
    subroutine integrations_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Integrations: execute tests*⟩
```
    end subroutine integrations_test
```

⟨*Integrations: public*⟩+≡
```
    public :: integrations_history_test
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_history_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Integrations: execute history tests⟩
      end subroutine integrations_history_test
```

Integration of test process

Compile and integrate an intrinsic test matrix element (`prc_test` type). The phase-space implementation is `phs_single` (single-particle phase space), the integrator is `mci_midpoint`.

The cross section for the $2 \rightarrow 2$ process $ss \rightarrow ss$ with its constant matrix element is given by

$$\sigma = c \times f \times \Phi_2 \times |M|^2. \tag{22.1}$$

c is the conversion constant

$$c = 0.3894 \times 10^{12} \text{ fb GeV}^2. \tag{22.2}$$

f is the flux of the incoming particles with mass $m = 125\,\text{GeV}$ and energy $\sqrt{s} = 1000\,\text{GeV}$

$$f = \frac{(2\pi)^4}{2\lambda^{1/2}(s, m^2, m^2)} = \frac{(2\pi)^4}{2\sqrt{s}\,\sqrt{s - 4m^2}} = 8.048 \times 10^{-4} \text{ GeV}^{-2} \tag{22.3}$$

Φ_2 is the volume of the two-particle phase space

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.5529 \times 10^{-5}. \tag{22.4}$$

The squared matrix element $|M|^2$ is unity. Combining everything, we obtain

$$\sigma = 8000 \text{ fb} \tag{22.5}$$

This number should appear as the final result.

Note: In this and the following test, we reset the Fortran compiler and flag variables immediately before they are printed, so the test is portable.

⟨*Integrations: execute tests*⟩≡
```
    call test (integrations_1, "integrations_1", &
        "intrinsic test process", &
        u, results)
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_1 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global

      write (u, "(A)")  "* Test output: integrations_1"
      write (u, "(A)")  "*   Purpose: integrate test process"
      write (u, "(A)")
```

```
      call syntax_model_file_init ()

      call global%global_init ()

      libname = "integration_1"
      procname = "prc_config_a"

      call prepare_test_library (global, libname, 1)
      call compile_library (libname, global)

      call var_list_set_string (global%var_list, var_str ("$run_id"), &
           var_str ("integrations1"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$phs_method"), &
           var_str ("single"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$integration_method"),&
           var_str ("midpoint"), is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?vis_history"),&
           .false., is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
           .false., is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("seed"), &
           0, is_known=.true.)

      call var_list_set_real (global%var_list, var_str ("sqrts"),&
           1000._default, is_known = .true.)

      call global%it_list%init ([1], [1000])

      call reset_interaction_counter ()
      call integrate_process (procname, global)

      call global%write (u, vars = [ &
           var_str ("$method"), &
           var_str ("sqrts"), &
           var_str ("$integration_method"), &
           var_str ("$phs_method"), &
           var_str ("$run_id")])

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: integrations_1"

    end subroutine integrations_1
```

Integration with cuts

Compile and integrate an intrinsic test matrix element (`prc_test` type) with cuts set.

⟨*Integrations: execute tests*⟩+≡
```
     call test (integrations_2, "integrations_2", &
         "intrinsic test process with cut", &
         u, results)
```
⟨*Integrations: tests*⟩+≡
```
     subroutine integrations_2 (u)
       integer, intent(in) :: u
       type(string_t) :: libname, procname
       type(rt_data_t), target :: global

       type(string_t) :: cut_expr_text
       type(ifile_t) :: ifile
       type(stream_t) :: stream
       type(parse_tree_t) :: parse_tree

       type(string_t), dimension(0) :: empty_string_array

       write (u, "(A)")  "* Test output: integrations_2"
       write (u, "(A)")  "*   Purpose: integrate test process with cut"
       write (u, "(A)")

       call syntax_model_file_init ()

       call global%global_init ()

       write (u, "(A)")  "* Prepare a cut expression"
       write (u, "(A)")

       call syntax_pexpr_init ()
       cut_expr_text = "all Pt > 100 [s]"
       call ifile_append (ifile, cut_expr_text)
       call stream_init (stream, ifile)
       call parse_tree_init_lexpr (parse_tree, stream, .true.)
       global%pn%cuts_lexpr => parse_tree_get_root_ptr (parse_tree)

       write (u, "(A)")  "* Build and initialize a test process"
       write (u, "(A)")

       libname = "integration_3"
       procname = "prc_config_a"

       call prepare_test_library (global, libname, 1)
       call compile_library (libname, global)

       call var_list_set_string (global%var_list, var_str ("$run_id"), &
           var_str ("integrations1"), is_known = .true.)
       call var_list_set_string (global%var_list, var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
       call var_list_set_string (global%var_list, var_str ("$phs_method"), &
           var_str ("single"), is_known = .true.)
       call var_list_set_string (global%var_list, var_str ("$integration_method"),&
           var_str ("midpoint"), is_known = .true.)
       call var_list_set_log (global%var_list, var_str ("?vis_history"),&
           .false., is_known = .true.)
```

```
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
           .false., is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("seed"), &
           0, is_known=.true.)

      call var_list_set_real (global%var_list, var_str ("sqrts"),&
           1000._default, is_known = .true.)

      call global%it_list%init ([1], [1000])

      call reset_interaction_counter ()
      call integrate_process (procname, global)

      call global%write (u, vars = empty_string_array)

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: integrations_2"

    end subroutine integrations_2
```

Standard phase space

Compile and integrate an intrinsic test matrix element (`prc_test` type) using
the default (`phs_wood`) phase-space implementation. We use an explicit phase-
space configuration file with a single channel and integrate by `mci_midpoint`.

⟨*Integrations: execute tests*⟩+≡
```
    call test (integrations_3, "integrations_3", &
         "standard phase space", &
         u, results)
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_3 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global
      integer :: u_phs

      write (u, "(A)")  "* Test output: integrations_3"
      write (u, "(A)")  "*   Purpose: integrate test process"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and parameters"
      write (u, "(A)")

      call syntax_model_file_init ()
      call syntax_phs_forest_init ()

      call global%global_init ()

      libname = "integration_3"
```

```
procname = "prc_config_a"

call prepare_test_library (global, libname, 1)
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("integrations1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("default"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?phs_s_mapping"),&
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known=.true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)

write (u, "(A)")  "* Create a scratch phase-space file"
write (u, "(A)")

u_phs = free_unit ()
open (u_phs, file = "integrations_3.phs", &
     status = "replace", action = "write")
call write_test_phs_file (u_phs, var_str ("prc_config_a_i1"))
close (u_phs)

call var_list_set_string (global%var_list, var_str ("$phs_file"),&
     var_str ("integrations_3.phs"), is_known = .true.)

call global%it_list%init ([1], [1000])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%write (u, vars = [ &
     var_str ("$phs_method"), &
     var_str ("$phs_file")])

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call syntax_phs_forest_final ()
```

```
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: integrations_3"

    end subroutine integrations_3
```

VAMP integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using
the single-channel (`phs_single`) phase-space implementation. The integration
method is `vamp`.

⟨*Integrations: execute tests*⟩+≡
```
      call test (integrations_4, "integrations_4", &
           "VAMP integration (one iteration)", &
           u, results)
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_4 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global

      write (u, "(A)")  "* Test output: integrations_4"
      write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and parameters"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()

      libname = "integrations_4_lib"
      procname = "integrations_4"

      call prepare_test_library (global, libname, 1, [procname])
      call compile_library (libname, global)

      call var_list_append_log (global%var_list, &
           var_str ("?rebuild_grids"), .true., intrinsic = .true.)

      call var_list_set_string (global%var_list, var_str ("$run_id"), &
           var_str ("r1"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$phs_method"), &
           var_str ("single"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$integration_method"),&
           var_str ("vamp"), is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
           .false., is_known = .true.)
```

```
      call var_list_set_log (global%var_list, var_str ("?vis_history"),&
           .false., is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
           .false., is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("seed"), &
           0, is_known=.true.)

      call var_list_set_real (global%var_list, var_str ("sqrts"),&
           1000._default, is_known = .true.)

      call global%it_list%init ([1], [1000])

      write (u, "(A)")  "* Integrate"
      write (u, "(A)")

      call reset_interaction_counter ()
      call integrate_process (procname, global)

      call global%pacify (efficiency_reset = .true., error_reset = .true.)
      call global%write (u, vars = [var_str ("$integration_method")], &
             pacify = .true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: integrations_4"

    end subroutine integrations_4
```

Multiple iterations integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the single-channel (`phs_single`) phase-space implementation. The integration method is `vamp`. We launch three iterations.

⟨*Integrations: execute tests*⟩+≡
```
      call test (integrations_5, "integrations_5", &
           "VAMP integration (three iterations)", &
           u, results)
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_5 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global

      write (u, "(A)")  "* Test output: integrations_5"
      write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
      write (u, "(A)")
```

2919

```
write (u, "(A)")  "* Initialize process and parameters"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()

libname = "integrations_5_lib"
procname = "integrations_5"

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known=.true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([3], [1000])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global)

call global%pacify (efficiency_reset = .true., error_reset = .true.)
call global%write (u, vars = [var_str ("$integration_method")], &
        pacify = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call syntax_model_file_final ()
```

```
   write (u, "(A)")
   write (u, "(A)")   "* Test output end: integrations_5"

end subroutine integrations_5
```

Multiple passes integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using
the single-channel (`phs_single`) phase-space implementation. The integration
method is `vamp`. We launch three passes with three iterations each.

⟨*Integrations: execute tests*⟩+≡
```
   call test (integrations_6, "integrations_6", &
        "VAMP integration (three passes)", &
        u, results)
```

⟨*Integrations: tests*⟩+≡
```
   subroutine integrations_6 (u)
     integer, intent(in) :: u
     type(string_t) :: libname, procname
     type(rt_data_t), target :: global
     type(string_t), dimension(0) :: no_vars

     write (u, "(A)")   "* Test output: integrations_6"
     write (u, "(A)")   "*   Purpose: integrate test process using VAMP"
     write (u, "(A)")

     write (u, "(A)")   "* Initialize process and parameters"
     write (u, "(A)")

     call syntax_model_file_init ()

     call global%global_init ()

     libname = "integrations_6_lib"
     procname = "integrations_6"

     call prepare_test_library (global, libname, 1, [procname])
     call compile_library (libname, global)

     call var_list_append_log (global%var_list, &
          var_str ("?rebuild_grids"), .true., intrinsic = .true.)

     call var_list_set_string (global%var_list, var_str ("$run_id"), &
          var_str ("r1"), is_known = .true.)
     call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known = .true.)
     call var_list_set_string (global%var_list, var_str ("$phs_method"), &
          var_str ("single"), is_known = .true.)
     call var_list_set_string (global%var_list, var_str ("$integration_method"),&
          var_str ("vamp"), is_known = .true.)
     call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
          .false., is_known = .true.)
     call var_list_set_log (global%var_list, var_str ("?vis_history"),&
```

```
          .false., is_known = .true.)
     call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
          .false., is_known = .true.)
     call var_list_set_int (global%var_list, var_str ("seed"), &
          0, is_known=.true.)

     call var_list_set_real (global%var_list, var_str ("sqrts"),&
          1000._default, is_known = .true.)

     call global%it_list%init ([3, 3, 3], [1000, 1000, 1000], &
          adapt = [.true., .true., .false.], &
          adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

     write (u, "(A)")  "* Integrate"
     write (u, "(A)")

     call reset_interaction_counter ()
     call integrate_process (procname, global)

     call global%pacify (efficiency_reset = .true., error_reset = .true.)
     call global%write (u, vars = no_vars, pacify = .true.)

     write (u, "(A)")
     write (u, "(A)")  "* Cleanup"

     call global%final ()
     call syntax_model_file_final ()

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: integrations_6"

   end subroutine integrations_6
```

VAMP and default phase space

Compile and integrate an intrinsic test matrix element (`prc_test` type) using
the default (`phs_wood`) phase-space implementation. The integration method
is `vamp`. We launch three passes with three iterations each. We enable channel
equivalences and groves.

⟨*Integrations: execute tests*⟩+≡
```
     call test (integrations_7, "integrations_7", &
          "VAMP integration with wood phase space", &
          u, results)
```

⟨*Integrations: tests*⟩+≡
```
   subroutine integrations_7 (u)
     integer, intent(in) :: u
     type(string_t) :: libname, procname
     type(rt_data_t), target :: global
     type(string_t), dimension(0) :: no_vars
     integer :: iostat, u_phs
     character(95) :: buffer
     type(string_t) :: phs_file
```

```
logical :: exist

write (u, "(A)")  "* Test output: integrations_7"
write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and parameters"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()

libname = "integrations_7_lib"
procname = "integrations_7"

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?phs_s_mapping"),&
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known=.true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([3, 3, 3], [1000, 1000, 1000], &
     adapt = [.true., .true., .false.], &
     adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

write (u, "(A)")  "* Integrate"
write (u, "(A)")
```

```
      call reset_interaction_counter ()
      call integrate_process (procname, global)

      call global%pacify (efficiency_reset = .true., error_reset = .true.)
      call global%write (u, vars = no_vars, pacify = .true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call global%final ()
      call syntax_phs_forest_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Generated phase-space file"
      write (u, "(A)")

      phs_file = procname // "_i1.r1.phs"
      inquire (file = char (phs_file), exist = exist)
      if (exist) then
         u_phs = free_unit ()
         open (u_phs, file = char (phs_file), action = "read", status = "old")
         iostat = 0
         do while (iostat == 0)
            read (u_phs, "(A)", iostat = iostat)  buffer
            if (iostat == 0)  write (u, "(A)")  trim (buffer)
         end do
         close (u_phs)
      else
         write (u, "(A)")  "[file is missing]"
      end if

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: integrations_7"

    end subroutine integrations_7
```

Structure functions

Compile and integrate an intrinsic test matrix element (`prc_test` type) using
the default (`phs_wood`) phase-space implementation. The integration method is
`vamp`. There is a structure function of type `unit_test`.

   We use a test structure function $f(x) = x$ for both beams. Together with the
$1/x_1 x_2$ factor from the phase-space flux and a unit matrix element, we should
get the same result as previously for the process without structure functions.
There is a slight correction due to the m_s mass which we set to zero here.

⟨*Integrations: execute tests*⟩+≡
```
    call test (integrations_8, "integrations_8", &
        "integration with structure function", &
        u, results)
```

⟨*Integrations: tests*⟩+≡

```
subroutine integrations_8 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global
  type(flavor_t) :: flv

  write (u, "(A)")  "* Test output: integrations_8"
  write (u, "(A)")  "*   Purpose: integrate test process using VAMP &
       &with structure function"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize process and parameters"
  write (u, "(A)")

  call syntax_model_file_init ()
  call syntax_phs_forest_init ()

  call global%global_init ()

  libname = "integrations_8_lib"
  procname = "integrations_8"

  call prepare_test_library (global, libname, 1, [procname])
  call compile_library (libname, global)

  call var_list_append_log (global%var_list, &
       var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
  call var_list_append_log (global%var_list, &
       var_str ("?rebuild_grids"), .true., intrinsic = .true.)

  call var_list_set_string (global%var_list, var_str ("$run_id"), &
       var_str ("r1"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$phs_method"), &
       var_str ("wood"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$integration_method"),&
       var_str ("vamp"), is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
       .true., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?vis_history"),&
       .false., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?phs_s_mapping"),&
       .false., is_known = .true.)
  call var_list_set_int (global%var_list, var_str ("seed"), &
       0, is_known=.true.)

  call var_list_set_real (global%var_list, var_str ("sqrts"),&
       1000._default, is_known = .true.)
  call var_list_set_real (global%var_list, var_str ("ms"), &
       0._default, is_known = .true.)
```

```
        call reset_interaction_counter ()

        call flavor_init (flv, 25, global%model)

        call global%beam_structure%init_sf (flavor_get_name ([flv, flv]), [1])
        call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

        write (u, "(A)")  "* Integrate"
        write (u, "(A)")

        call global%it_list%init ([1], [1000])
        call integrate_process (procname, global)

        call global%write (u, vars = [var_str ("ms")])

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call global%final ()
        call syntax_phs_forest_final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: integrations_8"

      end subroutine integrations_8
```

Integration history for VAMP integration with default phase space

This test is only run when event analysis can be done.

⟨*Integrations: execute history tests*⟩≡
```
    call test (integrations_history_1, "integrations_history_1", &
        "Test integration history files", &
        u, results)
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_history_1 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global
      type(string_t), dimension(0) :: no_vars
      integer :: iostat, u_his
      character(91) :: buffer
      type(string_t) :: his_file, ps_file, pdf_file
      logical :: exist, exist_ps, exist_pdf

      write (u, "(A)")  "* Test output: integrations_history_1"
      write (u, "(A)")  "*   Purpose: test integration history files"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and parameters"
      write (u, "(A)")
```

```
call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()

libname = "integrations_history_1_lib"
procname = "integrations_history_1"

call var_list_set_log (global%var_list, var_str ("?vis_history"), &
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?phs_s_mapping"),&
     .false., is_known = .true.)

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_real (global%var_list, var_str ("error_threshold"),&
     5E-6_default, is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known=.true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([2, 2, 2], [1000, 1000, 1000], &
     adapt = [.true., .true., .false.], &
     adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global, eff_reset = .true.)

call global%pacify (efficiency_reset = .true., error_reset = .true.)
call global%write (u, vars = no_vars, pacify = .true.)
```

```
   write (u, "(A)")
   write (u, "(A)")  "* Generated history files"
   write (u, "(A)")

   his_file = procname // ".r1-history.tex"
   ps_file  = procname // ".r1-history.ps"
   pdf_file = procname // ".r1-history.pdf"
   inquire (file = char (his_file), exist = exist)
   if (exist) then
      u_his = free_unit ()
      open (u_his, file = char (his_file), action = "read", status = "old")
      iostat = 0
      do while (iostat == 0)
         read (u_his, "(A)", iostat = iostat)  buffer
         if (iostat == 0)  write (u, "(A)")  trim (buffer)
      end do
      close (u_his)
   else
      write (u, "(A)")  "[History LaTeX file is missing]"
   end if
   inquire (file = char (ps_file), exist = exist_ps)
   if (exist_ps) then
      write (u, "(A)")  "[History Postscript file exists and is nonempty]"
   else
      write (u, "(A)")  "[History Postscript file is missing/non-regular]"
   end if
   inquire (file = char (pdf_file), exist = exist_pdf)
   if (exist_pdf) then
      write (u, "(A)")  "[History PDF file exists and is nonempty]"
   else
      write (u, "(A)")  "[History PDF file is missing/non-regular]"
   end if

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call global%final ()
   call syntax_phs_forest_final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: integrations_history_1"

 end subroutine integrations_history_1
```

22.8   Event Streams

This module manages I/O from/to multiple concurrent event streams. Usually, there is at most one input stream, but several output streams. For the latter, we set up an array which can hold `eio_t` (event I/O) objects of different dynamic types simultaneously. One of them may be marked as an input channel.

⟨event_streams.f90⟩≡
  ⟨*File header*⟩

  module event_streams

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use diagnostics !NODEP!
    use unit_tests
    use variables
    use models
    use processes
    use events
    use eio_data
    use eio_base
    use rt_data
    use dispatch

  ⟨*Standard module head*⟩

  ⟨*Event streams: public*⟩

  ⟨*Event streams: types*⟩

  contains

  ⟨*Event streams: tests*⟩

  ⟨*Event streams: procedures*⟩

  end module event_streams

22.8.1   Event Stream Array

Each entry is an `eio_t` object. Since the type is dynamic, we need a wrapper:

⟨*Event streams: types*⟩≡
```
    type :: event_stream_entry_t
       class(eio_t), allocatable :: eio
    end type event_stream_entry_t
```

An array of event-stream entry objects. If one of the entries is an input channel, `i_in` is the corresponding index.

⟨*Event streams: public*⟩≡
```
    public :: event_stream_array_t
```

⟨*Event streams: types*⟩+≡
```
    type :: event_stream_array_t
       type(event_stream_entry_t), dimension(:), allocatable :: entry
       integer :: i_in = 0
     contains
     ⟨Event streams: event stream array: TBP⟩
    end type event_stream_array_t
```

2929

Output.

⟨*Event streams: event stream array: TBP*⟩≡
```
    procedure :: write => event_stream_array_write
```

⟨*Event streams: procedures*⟩≡
```
    subroutine event_stream_array_write (object, unit)
      class(event_stream_array_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(1x,A)")  "Event stream array:"
      if (allocated (object%entry)) then
         select case (size (object%entry))
         case (0)
            write (u, "(3x,A)")  "[empty]"
         case default
            do i = 1, size (object%entry)
               if (i == object%i_in)  write (u, "(1x,A)")  "Input stream:"
               call object%entry(i)%eio%write (u)
            end do
         end select
      else
         write (u, "(3x,A)")  "[undefined]"
      end if
    end subroutine event_stream_array_write
```

Finalize all streams.

⟨*Event streams: event stream array: TBP*⟩+≡
```
    procedure :: final => event_stream_array_final
```

⟨*Event streams: procedures*⟩+≡
```
    subroutine event_stream_array_final (es_array)
      class(event_stream_array_t), intent(inout) :: es_array
      integer :: i
      do i = 1, size (es_array%entry)
         call es_array%entry(i)%eio%final ()
      end do
    end subroutine event_stream_array_final
```

Initialization. We use a generic `sample` name, open event I/O objects for all provided stream types (using the `dispatch_eio` routine), and initialize for the given list of process pointers. If there is an `input` argument, this channel is initialized as an input channel and appended to the array.

The `input_data` or, if not present, `data` may be modified. This happens if we open a stream for reading and get new information there.

⟨*Event streams: event stream array: TBP*⟩+≡
```
    procedure :: init => event_stream_array_init
```

⟨*Event streams: procedures*⟩+≡
```
    subroutine event_stream_array_init &
         (es_array, sample, stream_fmt, process_ptr, global, &
         data, input, input_sample, input_data, allow_switch, checkpoint, &
         error)
      class(event_stream_array_t), intent(out) :: es_array
```

2930

```
type(string_t), intent(in) :: sample
type(string_t), dimension(:), intent(in) :: stream_fmt
type(process_ptr_t), dimension(:), intent(in) :: process_ptr
type(rt_data_t), intent(in) :: global
type(event_sample_data_t), intent(inout), optional :: data
type(string_t), intent(in), optional :: input
type(string_t), intent(in), optional :: input_sample
type(event_sample_data_t), intent(inout), optional :: input_data
logical, intent(in), optional :: allow_switch
integer, intent(in), optional :: checkpoint
logical, intent(out), optional :: error
type(string_t) :: sample_in
integer :: n, i
logical :: success, switch
if (present (input)) then
   n = size (stream_fmt) + 1
else
   n = size (stream_fmt)
end if
if (present (input_sample)) then
   sample_in = input_sample
else
   sample_in = sample
end if
if (present (allow_switch)) then
   switch = allow_switch
else
   switch = .true.
end if
if (present (error)) then
   error = .false.
end if
if (present (checkpoint)) then
   allocate (es_array%entry (n + 1))
   call dispatch_eio &
        (es_array%entry(n+1)%eio, var_str ("checkpoint"), global)
   call es_array%entry(n+1)%eio%init_out (sample, process_ptr, data)
else
   allocate (es_array%entry (n))
end if
if (present (input)) then
   call dispatch_eio (es_array%entry(n)%eio, input, global)
   if (present (input_data)) then
      call es_array%entry(n)%eio%init_in &
           (sample_in, process_ptr, input_data, success)
   else
      call es_array%entry(n)%eio%init_in &
           (sample_in, process_ptr, data, success)
   end if
   if (success) then
      es_array%i_in = n
   else if (present (input_sample)) then
      if (present (error)) then
         error = .true.
```

```
            else
               call msg_fatal ("Events: &
                     &parameter mismatch in input, aborting")
            end if
         else
            call msg_message ("Events: &
                 &parameter mismatch, discarding old event set")
            call es_array%entry(n)%eio%final ()
            if (switch) then
               call msg_message ("Events: generating new events")
               call es_array%entry(n)%eio%init_out &
                     (sample, process_ptr, data)
            end if
         end if
      end if
      do i = 1, size (stream_fmt)
         call dispatch_eio (es_array%entry(i)%eio, stream_fmt(i), global)
         call es_array%entry(i)%eio%init_out (sample, process_ptr, data)
      end do
   end subroutine event_stream_array_init
```

Switch the (only) input channel to an output channel, so further events are appended to the respective stream.

⟨Event streams: event stream array: TBP⟩+≡
```
      procedure :: switch_inout => event_stream_array_switch_inout
```

⟨Event streams: procedures⟩+≡
```
   subroutine event_stream_array_switch_inout (es_array)
     class(event_stream_array_t), intent(inout) :: es_array
     integer :: n
     if (es_array%has_input ()) then
        n = es_array%i_in
        call es_array%entry(n)%eio%switch_inout ()
        es_array%i_in = 0
     else
        call msg_bug ("Reading events: switch_inout: no input stream selected")
     end if
   end subroutine event_stream_array_switch_inout
```

Output an event (with given process number) to all output streams. If there is no output stream, do nothing.

⟨Event streams: event stream array: TBP⟩+≡
```
      procedure :: output => event_stream_array_output
```

⟨Event streams: procedures⟩+≡
```
   subroutine event_stream_array_output (es_array, event, i_prc, event_index)
     class(event_stream_array_t), intent(inout) :: es_array
     type(event_t), intent(in), target :: event
     integer, intent(in) :: i_prc, event_index
     integer :: i
     do i = 1, size (es_array%entry)
        if (i /= es_array%i_in) then
           associate (eio => es_array%entry(i)%eio)
             if (eio%split) then
```

```
                  if (event_index > 1 .and. &
                      mod (event_index, eio%split_n_evt) == 1) then
                    call eio%split_out ()
                  end if
               end if
               call eio%output (event, i_prc, reading = es_array%i_in /= 0)
            end associate
         end if
      end do
   end subroutine event_stream_array_output
```

Input the `i_prc` index which selects the process for the current event. This is separated from reading the event, because it determines which event record to read. `iostat` may indicate an error or an EOF condition, as usual.

⟨*Event streams: event stream array: TBP*⟩+≡
```
    procedure :: input_i_prc => event_stream_array_input_i_prc
```

⟨*Event streams: procedures*⟩+≡
```
    subroutine event_stream_array_input_i_prc (es_array, i_prc, iostat)
      class(event_stream_array_t), intent(inout) :: es_array
      integer, intent(out) :: i_prc
      integer, intent(out) :: iostat
      integer :: n
      if (es_array%has_input ()) then
         n = es_array%i_in
         call es_array%entry(n)%eio%input_i_prc (i_prc, iostat)
      else
         call msg_fatal ("Reading events: no input stream selected")
      end if
    end subroutine event_stream_array_input_i_prc
```

Input an event from the selected input stream. `iostat` may indicate an error or an EOF condition, as usual.

⟨*Event streams: event stream array: TBP*⟩+≡
```
    procedure :: input_event => event_stream_array_input_event
```

⟨*Event streams: procedures*⟩+≡
```
    subroutine event_stream_array_input_event (es_array, event, iostat)
      class(event_stream_array_t), intent(inout) :: es_array
      type(event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      integer :: n
      if (es_array%has_input ()) then
         n = es_array%i_in
         call es_array%entry(n)%eio%input_event (event, iostat)
      else
         call msg_fatal ("Reading events: no input stream selected")
      end if
    end subroutine event_stream_array_input_event
```

Return true if there is an input channel among the event streams.

⟨*Event streams: event stream array: TBP*⟩+≡
```
    procedure :: has_input => event_stream_array_has_input
```

⟨*Event streams: procedures*⟩+≡
```
function event_stream_array_has_input (es_array) result (flag)
  class(event_stream_array_t), intent(in) :: es_array
  logical :: flag
  flag = es_array%i_in /= 0
end function event_stream_array_has_input
```

22.8.2   Unit Tests

⟨*Event streams: public*⟩+≡
```
public :: event_streams_test
```

⟨*Event streams: tests*⟩≡
```
subroutine event_streams_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*Event streams: execute tests*⟩
```
end subroutine event_streams_test
```

Empty event stream

This should set up an empty event output stream array, including initialization,
output, and finalization (which are all no-ops).

⟨*Event streams: execute tests*⟩≡
```
call test (event_streams_1, "event_streams_1", &
     "empty event stream array", &
     u, results)
```

⟨*Event streams: tests*⟩+≡
```
subroutine event_streams_1 (u)
  integer, intent(in) :: u
  type(event_stream_array_t) :: es_array
  type(rt_data_t) :: global
  type(event_t) :: event
  type(string_t) :: sample
  type(string_t), dimension(0) :: empty_string_array
  type(process_ptr_t), dimension(0) :: empty_process_ptr_array

  write (u, "(A)")  "* Test output: event_streams_1"
  write (u, "(A)")  "*   Purpose: handle empty event stream array"
  write (u, "(A)")

  sample = "event_streams_1"

  call es_array%init &
       (sample, empty_string_array, empty_process_ptr_array, global)
  call es_array%output (event, 42, 1)
  call es_array%write (u)
  call es_array%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: event_streams_1"
```

```
      end subroutine event_streams_1
```

Nontrivial event stream

Here we generate a trivial event and choose `raw` output as an entry in the stream
array.

⟨*Event streams: execute tests*⟩+≡
```
      call test (event_streams_2, "event_streams_2", &
           "nontrivial event stream array", &
           u, results)
```

⟨*Event streams: tests*⟩+≡
```
      subroutine event_streams_2 (u)
        integer, intent(in) :: u
        type(event_stream_array_t) :: es_array
        type(rt_data_t) :: global
        type(event_t), allocatable, target :: event
        type(process_t), allocatable, target :: process
        type(process_instance_t), allocatable, target :: process_instance
        type(process_ptr_t) :: process_ptr
        type(string_t) :: sample
        type(string_t), dimension(0) :: empty_string_array
        integer :: i_prc, iostat

        write (u, "(A)")  "* Test output: event_streams_2"
        write (u, "(A)")  "*   Purpose: handle empty event stream array"
        write (u, "(A)")

        call syntax_model_file_init ()
        call global%global_init ()

        write (u, "(A)")  "* Generate test process event"
        write (u, "(A)")

        allocate (process)
        process_ptr%ptr => process
        allocate (process_instance)
        call prepare_test_process (process, process_instance, global%model_list)
        call process_instance%setup_event_data ()

        allocate (event)
        call event%basic_init ()
        call event%connect (process_instance, process%get_model_ptr ())
        call event%generate (1, [0.4_default, 0.4_default])
        call event%evaluate_expressions ()
        call event%write (u)

        write (u, "(A)")
        write (u, "(A)") "* Allocate raw eio stream and write event to file"
        write (u, "(A)")

        sample = "event_streams_2"
```

2935

```
      call es_array%init (sample, [var_str ("raw")], [process_ptr], global)
      call es_array%output (event, 1, 1)
      call es_array%write (u)
      call es_array%final ()

      write (u, "(A)")
      write (u, "(A)") "* Reallocate raw eio stream for reading"
      write (u, "(A)")

      sample = "foo"
      call es_array%init (sample, empty_string_array, [process_ptr], global, &
           input = var_str ("raw"), input_sample = var_str ("event_streams_2"))
      call es_array%write (u)

      write (u, "(A)")
      write (u, "(A)") "* Reread event"
      write (u, "(A)")

      call es_array%input_i_prc (i_prc, iostat)

      write (u, "(1x,A,I0)")  "i_prc = ", i_prc
      write (u, "(A)")
      call es_array%input_event (event, iostat)
      call es_array%final ()

      call event%write (u)

      call global%final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: event_streams_2"

    end subroutine event_streams_2
```

Switch in/out

Here we generate an event file and test switching from writing to reading when
the file is exhausted.

⟨*Event streams: execute tests*⟩+≡
```
    call test (event_streams_3, "event_streams_3", &
         "switch input/output", &
         u, results)
```

⟨*Event streams: tests*⟩+≡
```
    subroutine event_streams_3 (u)
      integer, intent(in) :: u
      type(event_stream_array_t) :: es_array
      type(rt_data_t) :: global
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
```

2936

```
type(process_ptr_t) :: process_ptr
type(string_t) :: sample
type(string_t), dimension(0) :: empty_string_array
integer :: i_prc, iostat

write (u, "(A)")  "* Test output: event_streams_3"
write (u, "(A)")  "*   Purpose: handle in/out switching"
write (u, "(A)")

call syntax_model_file_init ()
call global%global_init ()

write (u, "(A)")  "* Generate test process event"
write (u, "(A)")

allocate (process)
process_ptr%ptr => process
allocate (process_instance)
call prepare_test_process (process, process_instance, global%model_list)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())
call event%generate (1, [0.4_default, 0.4_default])
call event%evaluate_expressions ()

write (u, "(A)") "* Allocate raw eio stream and write event to file"
write (u, "(A)")

sample = "event_streams_3"

call es_array%init (sample, [var_str ("raw")], [process_ptr], global)
call es_array%output (event, 1, 1)
call es_array%write (u)
call es_array%final ()

write (u, "(A)")
write (u, "(A)") "* Reallocate raw eio stream for reading"
write (u, "(A)")

call es_array%init (sample, empty_string_array, [process_ptr], global, &
     input = var_str ("raw"))
call es_array%write (u)

write (u, "(A)")
write (u, "(A)") "* Reread event"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
call es_array%input_event (event, iostat)

write (u, "(A)") "* Attempt to read another event (fail), then generate"
write (u, "(A)")
```

2937

```
call es_array%input_i_prc (i_prc, iostat)
if (iostat < 0) then
   call es_array%switch_inout ()
   call event%generate (1, [0.3_default, 0.3_default])
   call event%evaluate_expressions ()
   call es_array%output (event, 1, 2)
end if
call es_array%write (u)
call es_array%final ()

write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)") "* Reallocate raw eio stream for reading"
write (u, "(A)")

call es_array%init (sample, empty_string_array, [process_ptr], global, &
     input = var_str ("raw"))
call es_array%write (u)

write (u, "(A)")
write (u, "(A)") "* Reread two events and display 2nd event"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
call es_array%input_event (event, iostat)
call es_array%input_i_prc (i_prc, iostat)

call es_array%input_event (event, iostat)
call es_array%final ()

call event%write (u)

call global%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: event_streams_3"

end subroutine event_streams_3
```

Checksum

Here we generate an event file and repeat twice, once with identical parameters and once with modified parameters.

⟨*Event streams: execute tests*⟩+≡
```
call test (event_streams_4, "event_streams_4", &
     "check MD5 sum", &
     u, results)
```

⟨*Event streams: tests*⟩+≡
```
    subroutine event_streams_4 (u)
      integer, intent(in) :: u
      type(event_stream_array_t) :: es_array
      type(rt_data_t) :: global
      type(process_t), allocatable, target :: process
      type(process_ptr_t) :: process_ptr
      type(string_t) :: sample
      type(string_t), dimension(0) :: empty_string_array
      type(event_sample_data_t) :: data

      write (u, "(A)")  "* Test output: event_streams_4"
      write (u, "(A)")  "*   Purpose: handle in/out switching"
      write (u, "(A)")

      write (u, "(A)")  "* Generate test process event"
      write (u, "(A)")

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?check_event_file"), &
           .true., is_known = .true.)

      allocate (process)
      process_ptr%ptr => process

      write (u, "(A)") "* Allocate raw eio stream for writing"
      write (u, "(A)")

      sample = "event_streams_4"
      data%md5sum_cfg = "1234567890abcdef1234567890abcdef"

      call es_array%init &
           (sample, [var_str ("raw")], [process_ptr], global, data)
      call es_array%write (u)
      call es_array%final ()

      write (u, "(A)")
      write (u, "(A)") "* Reallocate raw eio stream for reading"
      write (u, "(A)")

      call es_array%init (sample, empty_string_array, [process_ptr], global, &
           data, input = var_str ("raw"))
      call es_array%write (u)
      call es_array%final ()

      write (u, "(A)")
      write (u, "(A)") "* Reallocate modified raw eio stream for reading (fail)"
      write (u, "(A)")

      data%md5sum_cfg = "1234567890_____1234567890_____"
      call es_array%init (sample, empty_string_array, [process_ptr], global, &
           data, input = var_str ("raw"))
      call es_array%write (u)
      call es_array%final ()
```

```
   write (u, "(A)")
   write (u, "(A)") "* Repeat ignoring checksum"
   write (u, "(A)")

   call var_list_set_log (global%var_list, var_str ("?check_event_file"), &
        .false., is_known = .true.)
   call es_array%init (sample, empty_string_array, [process_ptr], global, &
        data, input = var_str ("raw"))
   call es_array%write (u)
   call es_array%final ()

   call global%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: event_streams_4"

 end subroutine event_streams_4
```

22.9   Simulation

This module manages simulation: event generation and reading/writing of event files. The `simulation` object is intended to be used (via a pointer) outside of WHIZARD, if events are generated individually by an external driver.

⟨simulations.f90⟩≡
  ⟨*File header*⟩

  module simulations

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
  ⟨*Use file utils*⟩
    use limits, only: FMT_19 !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use sm_qcd
    use md5
    use ifiles
    use lexers
    use parser
    use variables
    use expressions
    use flavors
    use particles
    use state_matrices
    use interactions
    use models
    use beams
    use phs_forests
    use rng_base
    use selectors
```

```
    use prc_core
    use prclib_stacks
    use processes
    use events
    use event_transforms
    use decays
    use eio_data
    use eio_base
    use eio_raw
    use eio_ascii
    use rt_data
    use dispatch
    use process_configurations
    use compilations
    use integrations
    use event_streams
```

⟨*Standard module head*⟩

⟨*Simulations: public*⟩

⟨*Simulations: types*⟩

⟨*Simulations: interfaces*⟩

```
contains
```

⟨*Simulations: procedures*⟩

⟨*Simulations: tests*⟩

```
end module simulations
```

22.9.1   Event counting

In this object we collect statistical information about an event sample or sub-sample.

⟨*Simulations: types*⟩≡
```
    type :: counter_t
       integer :: total = 0
       integer :: generated = 0
       integer :: read = 0
       integer :: positive = 0
       integer :: negative = 0
       integer :: zero = 0
       integer :: excess = 0
       real(default) :: max_excess = 0
       real(default) :: sum_excess = 0
     contains
     ⟨Simulations: counter: TBP⟩
    end type counter_t
```

Output.

⟨Simulations: counter: TBP⟩≡
```
    procedure :: write => counter_write
```

⟨Simulations: procedures⟩≡
```
    subroutine counter_write (object, unit)
      class(counter_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
  1   format (3x,A,I0)
  2   format (5x,A,I0)
  3   format (5x,A,ES19.12)
      write (u, 1)  "Events total     = ", object%total
      write (u, 2)  "generated        = ", object%generated
      write (u, 2)  "read             = ", object%read
      write (u, 2)  "positive weight  = ", object%positive
      write (u, 2)  "negative weight  = ", object%negative
      write (u, 2)  "zero weight      = ", object%zero
      write (u, 2)  "excess weight    = ", object%excess
      if (object%excess /= 0) then
         write (u, 3)  "max excess       = ", object%max_excess
         write (u, 3)  "avg excess       = ", object%sum_excess / object%total
      end if
    end subroutine counter_write
```

This is a screen message: if there was an excess, display statistics.

⟨Simulations: counter: TBP⟩+≡
```
    procedure :: show_excess => counter_show_excess
```

⟨Simulations: procedures⟩+≡
```
    subroutine counter_show_excess (counter)
      class(counter_t), intent(in) :: counter
      if (counter%excess > 0) then
         write (msg_buffer, "(A,1x,I0,1x,A,1x,'(',F7.3,' %)')") &
               "Encountered events with excess weight:", counter%excess, &
               "events", 100 * counter%excess / real (counter%total)
         call msg_warning ()
         write (msg_buffer, "(A,ES10.3)") &
               "Maximum excess weight =", counter%max_excess
         call msg_message ()
         write (msg_buffer, "(A,ES10.3)") &
               "Average excess weight =", counter%sum_excess / counter%total
         call msg_message ()
      end if
    end subroutine counter_show_excess
```

Count an event. The weight and event source are optional; by default we assume
that the event has been generated and has positive weight.

⟨Simulations: counter: TBP⟩+≡
```
    procedure :: record => counter_record
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine counter_record (counter, weight, excess, from_file)
      class(counter_t), intent(inout) :: counter
      real(default), intent(in), optional :: weight, excess
      logical, intent(in), optional :: from_file
      counter%total = counter%total + 1
      if (present (from_file)) then
         if (from_file) then
            counter%read = counter%read + 1
         else
            counter%generated = counter%generated + 1
         end if
      else
         counter%generated = counter%generated + 1
      end if
      if (present (weight)) then
         if (weight > 0) then
            counter%positive = counter%positive + 1
         else if (weight < 0) then
            counter%negative = counter%negative + 1
         else
            counter%zero = counter%zero + 1
         end if
      else
         counter%positive = counter%positive + 1
      end if
      if (present (excess)) then
         if (excess > 0) then
            counter%excess = counter%excess + 1
            counter%max_excess = max (counter%max_excess, excess)
            counter%sum_excess = counter%sum_excess + excess
         end if
      end if
    end subroutine counter_record
```

22.9.2   Simulation: component sets

For each set of process components that share a MCI entry in the process
configuration, we keep a separate event record.

⟨*Simulations: types*⟩+≡
```
    type :: mci_set_t
      private
      integer :: n_components = 0
      integer, dimension(:), allocatable :: i_component
      type(string_t), dimension(:), allocatable :: component_id
      logical :: has_integral = .false.
      real(default) :: integral = 0
      real(default) :: error = 0
      real(default) :: weight_mci = 0
      type(counter_t) :: counter
    contains
    ⟨*Simulations: mci set: TBP*⟩
```

```
      end type mci_set_t
```

Output.

⟨*Simulations: mci set: TBP*⟩≡
```
    procedure :: write => mci_set_write
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine mci_set_write (object, unit)
      class(mci_set_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      write (u, "(3x,A)")  "Components:"
      do i = 1, object%n_components
         write (u, "(5x,I0,A,A,A)")  object%i_component(i), &
               ": '", char (object%component_id(i)), "'"
      end do
      if (object%has_integral) then
         write (u, "(3x,A," // FMT_19 // ")")  "Integral  = ", object%integral
         write (u, "(3x,A," // FMT_19 // ")")  "Error     = ", object%error
         write (u, "(3x,A,F13.10)")  "Weight    =", object%weight_mci
      else
         write (u, "(3x,A)")  "Integral  = [undefined]"
      end if
      call object%counter%write (u)
    end subroutine mci_set_write
```

Initialize: Get the indices and names for the process components that will contribute to this set.

⟨*Simulations: mci set: TBP*⟩+≡
```
    procedure :: init => mci_set_init
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine mci_set_init (object, i_mci, process)
      class(mci_set_t), intent(out) :: object
      integer, intent(in) :: i_mci
      type(process_t), intent(in), target :: process
      integer :: i
      call process%get_i_component (i_mci, object%i_component)
      object%n_components = size (object%i_component)
      allocate (object%component_id (object%n_components))
      do i = 1, size (object%component_id)
         object%component_id(i) = &
               process%get_component_id (object%i_component(i))
      end do
      if (process%has_integral (i_mci)) then
         object%integral = process%get_integral (i_mci)
         object%error = process%get_error (i_mci)
         object%has_integral = .true.
      end if
    end subroutine mci_set_init
```

22.9.3 Process-core Safe

This is an object that temporarily holds a process core object. We need this while rescanning a process with modified parameters. After the rescan, we want to restore the original state.

⟨*Simulations: types*⟩+≡
```
type :: core_safe_t
   class(prc_core_t), allocatable :: core
end type core_safe_t
```

22.9.4 Process Object

The simulation works on process objects. This subroutine makes a process object available for simulation. The process is in the process stack. If integration is not yet done, do it.

⟨*Simulations: procedures*⟩+≡
```
subroutine prepare_process (process, process_id, integrate, global)
  type(process_t), pointer, intent(out) :: process
  type(string_t), intent(in) :: process_id
  logical, intent(in) :: integrate
  type(rt_data_t), intent(inout), target :: global
  process => global%process_stack%get_process_ptr (process_id)
  if (.not. associated (process)) then
     if (integrate) then
        call msg_message ("Simulate: process '" &
             // char (process_id) // "' needs integration")
     else
        call msg_message ("Simulate: process '" &
             // char (process_id) // "' needs initialization")
     end if
     call integrate_process (process_id, global, init_only = .not. integrate)
     if (signal_is_pending ())  return
     process => global%process_stack%get_process_ptr (process_id)
     if (associated (process)) then
        if (integrate) then
           call msg_message ("Simulate: integration done")
           call global%process_stack%fill_result_vars (process_id)
        else
           call msg_message ("Simulate: process initialization done")
        end if
     else
        call msg_fatal ("Simulate: process '" &
             // char (process_id) // "' could not be initialized: aborting")
     end if
  end if
end subroutine prepare_process
```

22.9.5 Simulation entry

For each process that we consider for event generation, we need a separate entry. The entry separately records the process ID and run ID. The `weight_mci` array

is used for selecting a component set (which shares a MCI record inside the process container) when generating an event for the current process.

The simulation entry is an extension of the `event_t` event record. This core object contains configuration data, pointers to the process and process instance, the expressions, flags and values that are evaluated at runtime, and the resulting particle set.

The entry explicitly allocate the `process_instance`, which becomes the process-specific workspace for the event record.

If entries with differing environments are present simultaneously, we may need to switch QCD parameters and/or the model event by event. In this case, the `qcd` and/or `model` components are present.

⟨*Simulations: types*⟩+≡
```
type, extends (event_t) :: entry_t
   private
   type(string_t) :: process_id
   type(string_t) :: library
   type(string_t) :: run_id
   logical :: has_integral = .false.
   real(default) :: integral = 0
   real(default) :: error = 0
   real(default) :: process_weight = 0
   logical :: valid = .false.
   type(counter_t) :: counter
   integer :: n_in = 0
   integer :: n_mci = 0
   type(mci_set_t), dimension(:), allocatable :: mci_set
   type(selector_t) :: mci_selector
   type(core_safe_t), dimension(:), allocatable :: core_safe
   type(model_t), pointer :: model => null ()
   type(qcd_t) :: qcd
 contains
 ⟨Simulations: entry: TBP⟩
end type entry_t
```

Output. Write just the configuration, the event is written by a separate routine.

The `verbose` option is unused, it is required by the interface of the base-object method.

⟨*Simulations: entry: TBP*⟩≡
```
procedure :: write_config => entry_write_config
```

⟨*Simulations: procedures*⟩+≡
```
subroutine entry_write_config (object, unit)
  class(entry_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = output_unit (unit)
  write (u, "(3x,A,A,A)") "Process  = '", char (object%process_id), "'"
  write (u, "(3x,A,A,A)") "Library  = '", char (object%library), "'"
  write (u, "(3x,A,A,A)") "Run      = '", char (object%run_id), "'"
  write (u, "(3x,A,L1)")  "is valid = ", object%valid
  if (object%has_integral) then
     write (u, "(3x,A," // FMT_19 // ")") "Integral = ", object%integral
     write (u, "(3x,A," // FMT_19 // ")") "Error    = ", object%error
```

```
      write (u, "(3x,A,F13.10)")  "Weight     =", object%process_weight
   else
      write (u, "(3x,A)")  "Integral  = [undefined]"
   end if
   write (u, "(3x,A,I0)")   "MCI sets  = ", object%n_mci
   call object%counter%write (u)
   do i = 1, size (object%mci_set)
      write (u, "(A)")
      write (u, "(1x,A,I0,A)")  "MCI set #", i, ":"
      call object%mci_set(i)%write (u)
   end do
   if (allocated (object%core_safe)) then
      do i = 1, size (object%core_safe)
         write (u, "(1x,A,I0,A)")  "Saved process-component core #", i, ":"
         call object%core_safe(i)%core%write (u)
      end do
   end if
end subroutine entry_write_config
```

Finalizer. The `instance` pointer component of the `event_t` base type points to a target which we did explicitly allocate in the `entry_init` procedure. Therefore, we finalize and explicitly deallocate it here. Then we call the finalizer of the base type.

⟨*Simulations: entry: TBP*⟩+≡
```
   procedure :: final => entry_final
```

⟨*Simulations: procedures*⟩+≡
```
   subroutine entry_final (object)
     class(entry_t), intent(inout) :: object
     integer :: i
     if (associated (object%instance)) then
        do i = 1, object%n_mci
           call object%instance%final_simulation (i)
        end do
        call object%instance%final ()
        deallocate (object%instance)
     end if
     call object%event_t%final ()
   end subroutine entry_final
```

Initialization. Search for a process entry and allocate a process instance as an anonymous object, temporarily accessible via the `process_instance` pointer. Assign data by looking at the process object and at the environment.

If `n_alt` is set, we prepare for additional alternate sqme and weight entries.

If the process object is not found initially: if `integrate` is set, attempt an integration pass and try again. Otherwise, just initialize the object.

If `generate` is set, prepare the MCI objects for generating new events. For pure rescanning, this is not necessary.

When done, we assign the `instance` and `process` pointers of the base type by the `connect` method, so we can reference them later.

⟨*Simulations: entry: TBP*⟩+≡
```
   procedure :: init => entry_init
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_init (entry, process_id, integrate, generate, global, n_alt)
    class(entry_t), intent(inout), target :: entry
    type(string_t), intent(in) :: process_id
    logical, intent(in) :: integrate, generate
    type(rt_data_t), intent(inout), target :: global
    integer, intent(in), optional :: n_alt
    type(process_t), pointer :: process
    type(process_instance_t), pointer :: process_instance
    class(evt_t), pointer :: evt
    integer :: i
    logical :: enable_qcd

    enable_qcd = var_list_get_lval (global%var_list, var_str ("?ps_isr_active")) &
           .or. var_list_get_lval (global%var_list, var_str ("?ps_fsr_active")) &
           .or. var_list_get_lval (global%var_list, var_str &
                   ("?hadronization_active")) &
           .or. var_list_get_lval (global%var_list, var_str ("?mlm_matching")) &
           .or. var_list_get_lval (global%var_list, var_str ("?ckkw_matching")) &
           .or. var_list_get_lval (global%var_list, var_str ("?muli_active"))

    call prepare_process (process, process_id, integrate, global)
    if (signal_is_pending ())  return

    if (.not. process%has_matrix_element ()) then
       entry%has_integral = .true.
       entry%process_id = process_id
       entry%valid = .false.
       return
    end if

    call entry%basic_init (global%var_list, n_alt)

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data ()

    entry%process_id = process_id
    entry%library = process%get_library_name ()
    entry%run_id = process%get_run_id ()
    entry%n_in = process%get_n_in ()
    entry%n_mci = process%get_n_mci ()
    allocate (entry%mci_set (entry%n_mci))
    do i = 1, size (entry%mci_set)
       call entry%mci_set(i)%init (i, process)
    end do
    if (process%has_integral ()) then
       entry%integral = process%get_integral ()
       entry%error = process%get_error ()
       call entry%set_sigma (entry%integral)
       entry%has_integral = .true.
    end if

    call entry%set_selection (global%pn%selection_lexpr)
```

2948

```
call entry%set_reweight (global%pn%reweight_expr)
call entry%set_analysis (global%pn%analysis_lexpr)
if (generate) then
   do i = 1, entry%n_mci
      call process%prepare_simulation (i)
      call process_instance%init_simulation (i, entry%config%safety_factor)
   end do
end if

if (process%contains_unstable (global%model)) then
   call dispatch_evt_decay (evt, global)
   if (associated (evt))  call entry%import_transform (evt)
end if

if (enable_qcd) then
   call dispatch_evt_shower (evt, global, process)
   if (associated (evt))  call entry%import_transform (evt)
end if

call entry%connect (process_instance, global%model, global%process_stack)
call entry%setup_expressions ()
entry%model => process%get_model_ptr ()
call dispatch_qcd (entry%qcd, global)
entry%valid = .true.

end subroutine entry_init
```

Compute weights. The integral in the argument is the sum of integrals for all processes in the sample. After computing the process weights, we repeat the normalization procedure for the process components.

⟨*Simulations: entry: TBP*⟩+≡
```
procedure :: init_mci_selector => entry_init_mci_selector
```

⟨*Simulations: procedures*⟩+≡
```
subroutine entry_init_mci_selector (entry)
  class(entry_t), intent(inout) :: entry
  integer :: i
  if (entry%has_integral) then
     call entry%mci_selector%init (entry%mci_set%integral)
     do i = 1, entry%n_mci
        entry%mci_set(i)%weight_mci = entry%mci_selector%get_weight (i)
     end do
  end if
end subroutine entry_init_mci_selector
```

Select a MCI entry, using the embedded random-number generator.

⟨*Simulations: entry: TBP*⟩+≡
```
procedure :: select_mci => entry_select_mci
```

⟨*Simulations: procedures*⟩+≡
```
function entry_select_mci (entry) result (i_mci)
  class(entry_t), intent(inout) :: entry
  integer :: i_mci
  call entry%mci_selector%generate (entry%rng, i_mci)
```

```
      end function entry_select_mci
```

Record an event for this entry, i.e., increment the appropriate counters.

⟨*Simulations: entry: TBP*⟩+≡
```
    procedure :: record => entry_record
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine entry_record (entry, i_mci, from_file)
      class(entry_t), intent(inout) :: entry
      integer, intent(in) :: i_mci
      logical, intent(in), optional :: from_file
      real(default) :: weight, excess
      weight = entry%weight_prc
      excess = entry%excess_prc
      call entry%counter%record (weight, excess, from_file)
      call entry%mci_set(i_mci)%counter%record (weight, excess)
    end subroutine entry_record
```

Update and restore the process core that this entry accesses, when parameters change. If explicit arguments `model`, `qcd`, or `helicity_selection` are provided, use those. Otherwise use the parameters stored in the process object.

⟨*Simulations: entry: TBP*⟩+≡
```
    procedure :: update_process => entry_update_process
    procedure :: restore_process => entry_restore_process
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine entry_update_process (entry, model, qcd, helicity_selection)
      class(entry_t), intent(inout) :: entry
      type(model_t), intent(in), optional, target :: model
      type(qcd_t), intent(in), optional :: qcd
      type(helicity_selection_t), intent(in), optional :: helicity_selection
      type(process_t), pointer :: process
      class(prc_core_t), allocatable :: core
      integer :: i, n_components
      type(model_t), pointer :: model_local
      type(qcd_t) :: qcd_local
      if (present (model)) then
         model_local => model
      else
         model_local => entry%model
      end if
      if (present (qcd)) then
         qcd_local = qcd
      else
         qcd_local = entry%qcd
      end if
      process => entry%get_process_ptr ()
      n_components = process%get_n_components ()
      allocate (entry%core_safe (n_components))
      do i = 1, n_components
         if (process%has_matrix_element (i)) then
            call process%extract_component_core (i, core)
            call dispatch_core_update (core, &
                 model_local, helicity_selection, qcd_local, &
```

```
              entry%core_safe(i)%core)
          call process%restore_component_core (i, core)
        end if
      end do
    end subroutine entry_update_process

    subroutine entry_restore_process (entry)
      class(entry_t), intent(inout) :: entry
      type(process_t), pointer :: process
      class(prc_core_t), allocatable :: core
      integer :: i, n_components
      process => entry%get_process_ptr ()
      n_components = process%get_n_components ()
      do i = 1, n_components
        if (process%has_matrix_element (i)) then
          call process%extract_component_core (i, core)
          call dispatch_core_restore (core, entry%core_safe(i)%core)
          call process%restore_component_core (i, core)
        end if
      end do
      deallocate (entry%core_safe)
    end subroutine entry_restore_process
```

22.9.6 Entries for alternative environment

Entries for alternate environments. [No additional components anymore, so somewhat redundant.]

⟨*Simulations: types*⟩+≡
```
    type, extends (entry_t) :: alt_entry_t
     contains
     ⟨Simulations: alt entry: TBP⟩
    end type alt_entry_t
```

The alternative entries are there to re-evaluate the event, given momenta, in a different context.

Therefore, we allocate a local process object and use this as the reference for the local process instance, when initializing the entry. We temporarily import the `process` object into an `integration_t` wrapper, to take advantage of the associated methods. The local process object is built in the context of the current environment, here called `global`. Then, we initialize the process instance.

The `master_process` object contains the integration results to which we refer when recalculating an event. Therefore, we use this object instead of the locally built `process` when we extract the integration results.

The locally built `process` object should be finalized when done. It remains accessible via the `event_t` base object of `entry`, which contains pointers to the process and instance.

⟨*Simulations: alt entry: TBP*⟩≡
```
    procedure :: init_alt => alt_entry_init
```

⟨*Simulations: procedures*⟩+≡

```
      subroutine alt_entry_init (entry, process_id, &
         master_process, global)
    class(alt_entry_t), intent(inout), target :: entry
    type(string_t), intent(in) :: process_id
    type(process_t), intent(in), target :: master_process
    type(rt_data_t), intent(inout), target :: global
    class(rng_factory_t), allocatable :: rng_factory
    type(process_t), pointer :: process
    type(process_instance_t), pointer :: process_instance
    class(evt_t), pointer :: evt
    type(string_t) :: run_id
    type(integration_t) :: intg
    integer :: i
    logical :: enable_qcd

    enable_qcd = var_list_get_lval (global%var_list, var_str ("?ps_isr_active")) &
           .or. var_list_get_lval (global%var_list, var_str ("?ps_fsr_active")) &
           .or. var_list_get_lval (global%var_list, var_str &
                   ("?hadronization_active")) &
           .or. var_list_get_lval (global%var_list, var_str ("?mlm_matching")) &
           .or. var_list_get_lval (global%var_list, var_str ("?ckkw_matching")) &
           .or. var_list_get_lval (global%var_list, var_str ("?muli_active"))

    call msg_message ("Simulate: initializing alternate process setup ...")

    run_id = var_list_get_sval (global%var_list, var_str ("$run_id"))
    call var_list_set_log (global%var_list, var_str ("?rebuild_phase_space"), &
         .false., is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?check_phs_file"), &
         .false., is_known = .true.)
    call var_list_set_log (global%var_list, var_str ("?rebuild_grids"), &
         .false., is_known = .true.)

    call dispatch_qcd (entry%qcd, global)
    call dispatch_rng_factory (rng_factory, global)

    allocate (process)
    call process%init (process_id, run_id, global%prclib, &
         global%os_data, entry%qcd, rng_factory, global%model_list)
    call intg%setup_process (global, process)

    call entry%basic_init (global%var_list)

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data ()
    entry%process_id = process_id
    entry%library = process%get_library_name ()
    entry%run_id = run_id
    entry%n_mci = process%get_n_mci ()
    allocate (entry%mci_set (entry%n_mci))
    do i = 1, size (entry%mci_set)
       call entry%mci_set(i)%init (i, master_process)
```

2952

```
      end do
      if (master_process%has_integral ()) then
         entry%integral = master_process%get_integral ()
         entry%error = master_process%get_error ()
         call entry%set_sigma (entry%integral)
         entry%has_integral = .true.
      end if

      call entry%set_selection (global%pn%selection_lexpr)
      call entry%set_reweight (global%pn%reweight_expr)
      call entry%set_analysis (global%pn%analysis_lexpr)

      if (process%contains_unstable (global%model)) then
         call dispatch_evt_decay (evt, global)
         if (associated (evt))  call entry%import_transform (evt)
      end if

      if (enable_qcd) then
         call dispatch_evt_shower (evt, global)
         if (associated (evt))  call entry%import_transform (evt)
      end if

      call entry%connect (process_instance, global%model, global%process_stack)
      call entry%setup_expressions ()

      entry%model => process%get_model_ptr ()

      call msg_message ("...  alternate process setup complete.")

    end subroutine alt_entry_init
```

Copy the particle set from the master entry to the alternate entry. This is the
particle set of the hard process.

⟨*Simulations: alt entry: TBP*⟩+≡
```
    procedure :: fill_particle_set => entry_fill_particle_set
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine entry_fill_particle_set (alt_entry, entry)
      class(alt_entry_t), intent(inout) :: alt_entry
      class(entry_t), intent(in), target :: entry
      type(particle_set_t) :: pset
      call entry%get_particle_set_hard_proc (pset)
      call alt_entry%set_particle_set_hard_proc (pset)
      call particle_set_final (pset)
    end subroutine entry_fill_particle_set
```

22.9.7 The simulation type

Each simulation object corresponds to an event sample, identified by the `sample_id`.
    The simulation may cover several processes simultaneously. All process-
specific data, including the event records, are stored in the `entry` subobjects.

The `current` index indicates which record was selected last. `version` is foreseen to contain a tag on the `WHIZARD` event file version. It can be

⟨*Simulations: public*⟩≡
```
    public :: simulation_t
```

⟨*Simulations: types*⟩+≡
```
    type :: simulation_t
       private
       type(string_t) :: sample_id
       logical :: unweighted = .true.
       logical :: negative_weights = .false.
       integer :: norm_mode = NORM_UNDEFINED
       logical :: pacify = .false.
       integer :: n_max_tries = 10000
       integer :: n_prc = 0
       integer :: n_alt = 0
       logical :: has_integral = .false.
       logical :: valid
       real(default) :: integral = 0
       real(default) :: error = 0
       integer :: version = 1
       character(32) :: md5sum_prc = ""
       character(32) :: md5sum_cfg = ""
       character(32), dimension(:), allocatable :: md5sum_alt
       type(entry_t), dimension(:), allocatable :: entry
       type(alt_entry_t), dimension(:,:), allocatable :: alt_entry
       type(selector_t) :: process_selector
       integer :: n_evt_requested = 0
       integer :: split_n_evt = 0
       integer :: split_index = 0
       type(counter_t) :: counter
       class(rng_t), allocatable :: rng
       integer :: i_prc = 0
       integer :: i_mci = 0
       real(default) :: weight = 0
       real(default) :: excess = 0
     contains
     ⟨Simulations: simulation: TBP⟩
    end type simulation_t
```

Output. `write_config` writes just the configuration. `write` as a method of the base type `event_t` writes the current event and process instance, depending on options.

⟨*Simulations: simulation: TBP*⟩≡
```
    procedure :: write => simulation_write
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_write (object, unit)
      class(simulation_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u, i
      u = output_unit (unit)
      call write_separator_double (u)
      write (u, "(1x,A,A,A)")  "Event sample: '", char (object%sample_id), "'"
```

```
write (u, "(3x,A,I0)")  "Processes    = ", object%n_prc
if (object%n_alt > 0) then
   write (u, "(3x,A,I0)")  "Alt.wgts     = ", object%n_alt
end if
write (u, "(3x,A,L1)")  "Unweighted   = ", object%unweighted
write (u, "(3x,A,A)")   "Event norm   = ", &
     char (event_normalization_string (object%norm_mode))
write (u, "(3x,A,L1)")  "Neg. weights = ", object%negative_weights
write (u, "(3x,A,L1)")  "Pacify       = ", object%pacify
write (u, "(3x,A,I0)")  "Max. tries   = ", object%n_max_tries
if (object%has_integral) then
   write (u, "(3x,A," // FMT_19 // ")")  "Integral     = ", object%integral
   write (u, "(3x,A," // FMT_19 // ")")  "Error        = ", object%error
else
   write (u, "(3x,A)")  "Integral     = [undefined]"
end if
write (u, "(3x,A,L1)")  "Sim. valid   = ", object%valid
write (u, "(3x,A,I0)")  "Ev.file ver. = ", object%version
if (object%md5sum_prc /= "") then
   write (u, "(3x,A,A,A)")  "MD5 sum (proc)  = '", object%md5sum_prc, "'"
end if
if (object%md5sum_cfg /= "") then
   write (u, "(3x,A,A,A)")  "MD5 sum (config) = '", object%md5sum_cfg, "'"
end if
write (u, "(3x,A,I0)")  "Events requested  = ", object%n_evt_requested
if (object%split_n_evt > 0) then
   write (u, "(3x,A,I0)")  "Events per file   = ", object%split_n_evt
   write (u, "(3x,A,I0)")  "First file index  = ", object%split_index
end if
call object%counter%write (u)
call write_separator (u)
if (object%i_prc /= 0) then
   write (u, "(1x,A)")  "Current event:"
   write (u, "(3x,A,I0,A,A)")  "Process #", &
        object%i_prc, ": ", &
        char (object%entry(object%i_prc)%process_id)
   write (u, "(3x,A,I0)")  "MCI set #", object%i_mci
   write (u, "(3x,A," // FMT_19 // ")")  "Weight   = ", object%weight
   if (object%excess /= 0) &
        write (u, "(3x,A," // FMT_19 // ")")  "Excess   = ", object%excess
else
   write (u, "(1x,A,I0,A,A)")  "Current event: [undefined]"
end if
call write_separator (u)
if (allocated (object%rng)) then
   call object%rng%write (u)
else
   write (u, "(3x,A)")  "Random-number generator: [undefined]"
end if
if (allocated (object%entry)) then
   do i = 1, size (object%entry)
      if (i == 1) then
         call write_separator_double (u)
      else
```

```
          call write_separator (u)
        end if
        write (u, "(1x,A,I0,A)") "Process #", i, ":"
        call object%entry(i)%write_config (u)
      end do
    end if
    call write_separator_double (u)
  end subroutine simulation_write
```

Write the current event record. If an explicit index is given, write that event record.

We implement writing to `unit` (event contents / debugging format) and writing to an `eio` event stream (storage). We include a `testflag` in order to suppress numerical noise in the testsuite.

⟨*Simulations: simulation: TBP*⟩+≡
```
    generic :: write_event => write_event_unit
    procedure :: write_event_unit => simulation_write_event_unit
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_write_event_unit (object, unit, i_prc, verbose, testflag)
      class(simulation_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer, intent(in), optional :: i_prc
      logical, intent(in), optional :: testflag
      integer :: current
      if (present (i_prc)) then
         current = i_prc
      else
         current = object%i_prc
      end if
      if (current > 0) then
         call object%entry(current)%write (unit, verbose = verbose, &
              testflag = testflag)
      else
         call msg_fatal ("Simulation: write event: no process selected")
      end if
    end subroutine simulation_write_event_unit
```

This writes one of the alternate events, if allocated.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: write_alt_event => simulation_write_alt_event
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_write_alt_event (object, unit, j_alt, i_prc, &
        verbose, testflag)
      class(simulation_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer, intent(in), optional :: j_alt
      integer, intent(in), optional :: i_prc
      logical, intent(in), optional :: verbose
      logical, intent(in), optional :: testflag
      integer :: i, j
```

```
            if (present (j_alt)) then
               j = j_alt
            else
               j = 1
            end if
            if (present (i_prc)) then
               i = i_prc
            else
               i = object%i_prc
            end if
            if (i > 0) then
               if (j> 0 .and. j <= object%n_alt) then
                  call object%alt_entry(i,j)%write (unit, verbose = verbose, &
                       testflag = testflag)
               else
                  call msg_fatal ("Simulation: write alternate event: out of range")
               end if
            else
               call msg_fatal ("Simulation: write alternate event: no process selected")
            end if
      end subroutine simulation_write_alt_event
```

Finalizer.

⟨Simulations: simulation: TBP⟩+≡
```
      procedure :: final => simulation_final
```

⟨Simulations: procedures⟩+≡
```
      subroutine simulation_final (object)
        class(simulation_t), intent(inout) :: object
        integer :: i, j
        if (allocated (object%entry)) then
           do i = 1, size (object%entry)
              call object%entry(i)%final ()
           end do
        end if
        if (allocated (object%alt_entry)) then
           do j = 1, size (object%alt_entry, 2)
              do i = 1, size (object%alt_entry, 1)
                 call object%alt_entry(i,j)%final ()
              end do
           end do
        end if
        if (allocated (object%rng))  call object%rng%final ()
      end subroutine simulation_final
```

Initialization. We can deduce all data from the given list of process IDs and the global data set. The process objects are taken from the stack. Once the individual integrals are known, we add them (and the errors), to get the sample integral.

  If there are alternative environments, we suspend initialization for setting up alternative process objects, then restore the master process and its parameters. The generator or rescanner can then switch rapidly between processes.

If `integrate` is set, we make sure that all affected processes are integrated before simulation. This is necessary if we want to actually generate events. If `integrate` is unset, we don't need the integral because we just rescan existing events. In that case, we just need compiled matrix elements.

If `generate` is set, we prepare for actually generating events. Otherwise, we may only read and rescan events.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: init => simulation_init
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_init &
    (simulation, process_id, integrate, generate, global, alt_env)
  class(simulation_t), intent(out), target :: simulation
  type(string_t), dimension(:), intent(in) :: process_id
  logical, intent(in) :: integrate, generate
  type(rt_data_t), intent(inout), target :: global
  type(rt_data_t), dimension(:), intent(inout), target, optional :: alt_env
  class(rng_factory_t), allocatable :: rng_factory
  type(string_t) :: norm_string, version_string
  integer :: i, j
  simulation%sample_id = var_list_get_sval (global%var_list, &
      var_str ("$sample"))
  simulation%unweighted = var_list_get_lval (global%var_list, &
      var_str ("?unweighted"))
  simulation%negative_weights = var_list_get_lval (global%var_list, &
      var_str ("?negative_weights"))
  version_string = var_list_get_sval (global%var_list, &
      var_str ("$event_file_version"))
  norm_string = var_list_get_sval (global%var_list, &
      var_str ("$sample_normalization"))
  simulation%norm_mode = &
      event_normalization_mode (norm_string, simulation%unweighted)
  simulation%pacify = var_list_get_lval (global%var_list, &
      var_str ("?sample_pacify"))
  simulation%n_max_tries = var_list_get_ival (global%var_list, &
      var_str ("sample_max_tries"))
  simulation%split_n_evt = var_list_get_ival (global%var_list, &
      var_str ("sample_split_n_evt"))
  simulation%split_index = var_list_get_ival (global%var_list, &
      var_str ("sample_split_index"))
  select case (size (process_id))
  case (0)
     call msg_error ("Simulation: no process selected")
  case (1)
     write (msg_buffer, "(A,A,A)") &
           "Starting simulation for process '", &
           char (process_id(1)), "'"
     call msg_message ()
  case default
     write (msg_buffer, "(A,A,A)") &
           "Starting simulation for processes '", &
           char (process_id(1)), "' etc."
     call msg_message ()
  end select
```

2958

```
select case (char (version_string))
case ("2.0","2.1")
   simulation%version = 0
case default
   simulation%version = 1
end select
if (simulation%version == 0) then
   call msg_fatal ("Event file formats older than version 2.2 are " &
      // "not compatible with this version.")
end if
simulation%n_prc = size (process_id)
allocate (simulation%entry (simulation%n_prc))
if (present (alt_env)) then
   simulation%n_alt = size (alt_env)
   do i = 1, simulation%n_prc
      call simulation%entry(i)%init &
         (process_id(i), integrate, generate, global, simulation%n_alt)
      if (signal_is_pending ())  return
   end do
   if (.not. any (simulation%entry%valid)) then
      call msg_error ("Simulate: no process has a valid matrix element.")
      simulation%valid = .false.
      return
   end if
   call simulation%update_processes ()
   allocate (simulation%alt_entry (simulation%n_prc, simulation%n_alt))
   allocate (simulation%md5sum_alt (simulation%n_alt))
   simulation%md5sum_alt = ""
   do j = 1, simulation%n_alt
      do i = 1, simulation%n_prc
         call simulation%alt_entry(i,j)%init_alt (process_id(i), &
            simulation%entry(i)%get_process_ptr (), alt_env(j))
         if (signal_is_pending ())  return
      end do
   end do
   call simulation%restore_processes ()
else
   do i = 1, simulation%n_prc
      call simulation%entry(i)%init &
         (process_id(i), integrate, generate, global)
      if (signal_is_pending ())  return
   end do
   if (.not. any (simulation%entry%valid)) then
      call msg_error ("Simulate: " &
         // "no process has a valid matrix element.")
      simulation%valid = .false.
      return
   end if
end if
call dispatch_rng_factory (rng_factory, global)
call rng_factory%make (simulation%rng)
if (all (simulation%entry%has_integral)) then
   simulation%integral = sum (simulation%entry%integral)
   simulation%error = sqrt (sum (simulation%entry%error ** 2))
```

```
              simulation%has_integral = .true.
              if (integrate .and. generate) then
                 do i = 1, simulation%n_prc
                    if (simulation%entry(i)%integral < 0 .and. .not. &
                         simulation%negative_weights) then
                       call msg_fatal ("Integral of process '" // &
                            char (process_id (i)) // "'is negative.")
                    end if
                 end do
              end if
           else
              if (integrate .and. generate) &
                   call msg_error ("Simulation contains undefined integrals.")
           end if
           if (simulation%integral > 0 .or. &
                (simulation%integral < 0 .and. simulation%negative_weights)) then
              simulation%valid = .true.
           else if (generate) then
              call msg_error ("Simulate: " &
                   // "sum of process integrals must be positive; skipping.")
              simulation%valid = .false.
              return
           end if
           if (simulation%valid)  call simulation%compute_md5sum ()
         end subroutine simulation_init
```

The number of events that we want to simulate is determined by the settings
of `n_events`, `luminosity`, and `?unweighted`. For weighted events, we take
`n_events` at face value as the number of matrix element calls. For unweighted
events, if the process is a decay, `n_events` is the number of unweighted events.
In these cases, the luminosity setting is ignored.

For unweighted events with a scattering process, we calculate the event num-
ber that corresponds to the luminosity, given the current value of the integral.
We then compare this with `n_events` and choose the larger number.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: compute_n_events => simulation_compute_n_events
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_compute_n_events (simulation, n_events, var_list)
      class(simulation_t), intent(in) :: simulation
      integer, intent(out) :: n_events
      type(var_list_t) :: var_list
      real(default) :: lumi, x_events_lumi
      integer :: n_events_lumi
      logical :: is_scattering
      n_events = var_list_get_ival (var_list, var_str ("n_events"))
      lumi = var_list_get_rval (var_list, var_str ("luminosity"))
      if (simulation%unweighted) then
         is_scattering = simulation%entry(1)%n_in == 2
         if (is_scattering) then
            x_events_lumi = abs (simulation%integral * lumi)
            if (x_events_lumi < huge (n_events)) then
               n_events_lumi = nint (x_events_lumi)
```

```
          else
             call msg_message ("Simulation: luminosity too large, &
                  &limiting number of events")
             n_events_lumi = huge (n_events)
          end if
          if (n_events_lumi > n_events) then
             call msg_message ("Simulation: using n_events as computed from &
                  &luminosity value")
             n_events = n_events_lumi
          else
             write (msg_buffer, "(A,1x,I0)") &
                  "Simulation: requested number of events =", n_events
             call msg_message ()
             write (msg_buffer, "(A,1x,ES11.4)") &
                  "              corr. to luminosity [fb-1] = ", &
                   n_events / simulation%integral
             call msg_message ()
          end if
       end if
    end if
  end subroutine simulation_compute_n_events
```

Compute the checksum of the process set. We retrieve the MD5 sums of all processes. This depends only on the process definitions, while parameters are not considered. The configuration checksum is retrieved from the MCI records in the process objects and furthermore includes beams, parameters, integration results, etc., so matching the latter should guarantee identical physics.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: compute_md5sum => simulation_compute_md5sum
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_compute_md5sum (simulation)
      class(simulation_t), intent(inout) :: simulation
      type(process_t), pointer :: process
      type(string_t) :: buffer
      integer :: j, i, n_mci, i_mci, n_component, i_component
      if (simulation%md5sum_prc == "") then
         buffer = ""
         do i = 1, simulation%n_prc
            if (.not. simulation%entry(i)%valid) cycle
            process => simulation%entry(i)%get_process_ptr ()
            n_component = process%get_n_components ()
            do i_component = 1, n_component
               if (process%has_matrix_element (i_component)) then
                  buffer = buffer // process%get_md5sum_prc (i_component)
               end if
            end do
         end do
         simulation%md5sum_prc = md5sum (char (buffer))
      end if
      if (simulation%md5sum_cfg == "") then
         buffer = ""
         do i = 1, simulation%n_prc
            if (.not. simulation%entry(i)%valid) cycle
```

```
         process => simulation%entry(i)%get_process_ptr ()
         n_mci = process%get_n_mci ()
         do i_mci = 1, n_mci
            buffer = buffer // process%get_md5sum_mci (i_mci)
         end do
      end do
      simulation%md5sum_cfg = md5sum (char (buffer))
   end if
   do j = 1, simulation%n_alt
      if (simulation%md5sum_alt(j) == "") then
         buffer = ""
         do i = 1, simulation%n_prc
            process => simulation%alt_entry(i,j)%get_process_ptr ()
            buffer = buffer // process%get_md5sum_cfg ()
         end do
         simulation%md5sum_alt(j) = md5sum (char (buffer))
      end if
   end do
 end subroutine simulation_compute_md5sum
```

Initialize the process selector, using the entry integrals as process weights.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: init_process_selector => simulation_init_process_selector
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_init_process_selector (simulation)
      class(simulation_t), intent(inout) :: simulation
      integer :: i
      if (simulation%has_integral) then
         call simulation%process_selector%init (simulation%entry%integral)
         do i = 1, simulation%n_prc
            associate (entry => simulation%entry(i))
              if (.not. entry%valid) then
                 call msg_warning ("Process '" // char (entry%process_id) // &
                      "': matrix element vanishes, no events can be generated.")
                 cycle
              end if
              call entry%init_mci_selector ()
              entry%process_weight = simulation%process_selector%get_weight (i)
            end associate
         end do
      end if
    end subroutine simulation_init_process_selector
```

Select a process, using the random-number generator.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: select_prc => simulation_select_prc
```

⟨*Simulations: procedures*⟩+≡
```
    function simulation_select_prc (simulation) result (i_prc)
      class(simulation_t), intent(inout) :: simulation
      integer :: i_prc
      call simulation%process_selector%generate (simulation%rng, i_prc)
    end function simulation_select_prc
```

2962

Select a MCI set for the selected process.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: select_mci => simulation_select_mci
```

⟨*Simulations: procedures*⟩+≡
```
    function simulation_select_mci (simulation) result (i_mci)
      class(simulation_t), intent(inout) :: simulation
      integer :: i_mci
      if (simulation%i_prc /= 0) then
          i_mci = simulation%entry(simulation%i_prc)%select_mci ()
      end if
    end function simulation_select_mci
```

Generate a predefined number of events. First select a process and a component set, then generate an event for that process and factorize the quantum state. The pair of random numbers can be used for factorization.

When generating events, we drop all configurations where the event is marked as incomplete. This happens if the event fails cuts. In fact, such events are dropped already by the sampler if unweighting is in effect, so this can happen only for weighted events. By setting a limit given by `sample_max_tries` (user parameter), we can avoid an endless loop.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: generate => simulation_generate
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_generate (simulation, n, es_array)
      class(simulation_t), intent(inout) :: simulation
      integer, intent(in) :: n
      type(event_stream_array_t), intent(inout), optional :: es_array
      type(string_t) :: str1, str2, str3
      logical :: generate_new
      integer :: i, j
      simulation%n_evt_requested = n
      call simulation%entry%set_n (n)
      if (simulation%n_alt > 0)  call simulation%alt_entry%set_n (n)
      str1 = "Events: generating"
      if (present (es_array)) then
          if (es_array%has_input ())  str1 = "Events: reading"
      end if
      if (simulation%entry(1)%config%unweighted) then
          str2 = "unweighted"
      else
          str2 = "weighted"
      end if
      if (simulation%entry(1)%config%factorization_mode == &
          FM_IGNORE_HELICITY) then
          str3 = ", unpolarized"
      else
          str3 = ", polarized"
      end if
      write (msg_buffer, "(A,1x,I0,1x,A,1x,A)")  char (str1), n, &
          char (str2) // char(str3), "events ..."
```

```
call msg_message ()
write (msg_buffer, "(A,1x,A)") "Events: event normalization mode", &
    char (event_normalization_string (simulation%norm_mode))
call msg_message ()
do i = 1, n
   if (present (es_array)) then
      call simulation%read_event (es_array, .true., generate_new)
   else
      generate_new = .true.
   end if
   if (generate_new) then
      simulation%i_prc = simulation%select_prc ()
      simulation%i_mci = simulation%select_mci ()
      associate (entry => simulation%entry(simulation%i_prc))
        do j = 1, simulation%n_max_tries
           if (.not. entry%valid)  call msg_warning &
                 ("Process '" // char (entry%process_id) // "': " // &
                 "matrix element vanishes, no events can be generated.")
           call entry%generate (simulation%i_mci)
           if (signal_is_pending ()) return
           if (entry%particle_set_exists)  exit
        end do
        if (.not. entry%particle_set_exists) then
           write (msg_buffer, "(A,I0,A)")  "Simulation: failed to &
                &generate valid event after ", &
                simulation%n_max_tries, " tries (sample_max_tries)"
           call msg_fatal ()
        end if
        call entry%evaluate_expressions ()
        if (signal_is_pending ()) return
        simulation%weight = entry%weight_ref
        simulation%excess = entry%excess_prc
        call simulation%counter%record &
             (simulation%weight, simulation%excess)
        call entry%record (simulation%i_mci)
      end associate
   else
      associate (entry => simulation%entry(simulation%i_prc))
        call entry%accept_sqme_ref ()
        call entry%accept_weight_ref ()
        call entry%check ()
        call entry%evaluate_expressions ()
        if (signal_is_pending ()) return
        simulation%weight = entry%weight_ref
        simulation%excess = entry%excess_prc
        call simulation%counter%record &
             (simulation%weight, simulation%excess, from_file=.true.)
        call entry%record (simulation%i_mci, from_file=.true.)
      end associate
   end if
   call simulation%calculate_alt_entries ()
   if (signal_is_pending ()) return
   if (simulation%pacify)  call pacify (simulation)
   if (present (es_array)) then
```

```
        call simulation%write_event (es_array)
      end if
    end do
    call msg_message ("        ... event sample complete.")
    call simulation%counter%show_excess ()
  end subroutine simulation_generate
```

Compute the event matrix element and weight for all alternative environments, given the current event and selected process. We first copy the particle set, then temporarily update the process core with local parameters, recalculate everything, and restore the process core.

The event weight is obtained by rescaling the original event weight with the ratio of the new and old `sqme` values. (In particular, if the old value was zero, the weight will stay zero.)

Note: this may turn out to be inefficient because we always replace all parameters and recalculate everything, once for each event and environment. However, a more fine-grained control requires more code. In any case, while we may keep multiple process cores (which stay constant for a simulation run), we still have to update the external matrix element parameters event by event. The matrix element "object" is present only once.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: calculate_alt_entries => simulation_calculate_alt_entries
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_calculate_alt_entries (simulation)
      class(simulation_t), intent(inout) :: simulation
      real(default) :: factor
      real(default), dimension(:), allocatable :: sqme_alt, weight_alt
      integer :: n_alt, i, j
      i = simulation%i_prc
      n_alt = simulation%n_alt
      if (n_alt == 0)  return
      allocate (sqme_alt (n_alt), weight_alt (n_alt))
      associate (entry => simulation%entry(i))
        do j = 1, n_alt
          if (signal_is_pending ())  return
          factor = entry%get_kinematical_weight ()
          associate (alt_entry => simulation%alt_entry(i,j))
            call alt_entry%update_process ()
            call alt_entry%select &
                 (entry%get_i_mci (), entry%get_i_term (), entry%get_channel ())
            call alt_entry%fill_particle_set (entry)
            call alt_entry%recalculate &
                 (update_sqme = .true., weight_factor = factor)
            if (signal_is_pending ())  return
            call alt_entry%accept_sqme_prc ()
            call alt_entry%update_normalization ()
            call alt_entry%accept_weight_prc ()
            call alt_entry%check ()
            call alt_entry%evaluate_expressions ()
            if (signal_is_pending ())  return
            call alt_entry%restore_process ()
            sqme_alt(j) = alt_entry%sqme_ref
```

```
        weight_alt(j) = alt_entry%weight_ref
      end associate
    end do
    call entry%set (sqme_alt = sqme_alt, weight_alt = weight_alt)
    call entry%check ()
    call entry%store_alt_values ()
  end associate
end subroutine simulation_calculate_alt_entries
```

Rescan an undefined number of events.

If `update_event` or `update_sqme` is set, we have to recalculate the event, starting from the particle set. If the latter is set, this includes the squared matrix element (i.e., the amplitude is evaluated). Otherwise, only kinematics and observables derived from it are recovered.

If any of the update flags is set, we will come up with separate `sqme_prc` and `weight_prc` values. (The latter is only distinct if `update_weight` is set.) Otherwise, we accept the reference values.

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: rescan => simulation_rescan
```

⟨*Simulations: procedures*⟩+≡
```
subroutine simulation_rescan &
    (simulation, n, es_array, update_event, update_sqme, update_weight, &
     recover_beams, global)
  class(simulation_t), intent(inout) :: simulation
  integer, intent(in) :: n
  type(event_stream_array_t), intent(inout) :: es_array
  logical, intent(in) :: update_event, update_sqme, update_weight
  logical, intent(in) :: recover_beams
  type(rt_data_t), intent(inout) :: global
  type(qcd_t) :: qcd
  type(string_t) :: str1, str2, str3
  logical :: complete
  str1 = "Rescanning"
  if (simulation%entry(1)%config%unweighted) then
     str2 = "unweighted"
  else
     str2 = "weighted"
  end if
  simulation%n_evt_requested = n
  call simulation%entry%set_n (n)
  if (update_sqme .or. update_weight) then
     call dispatch_qcd (qcd, global)
     call simulation%update_processes &
         (global%model, qcd, global%get_helicity_selection ())
     str3 = "(process parameters updated) "
  else
     str3 = ""
  end if
  write (msg_buffer, "(A,1x,A,1x,A,A,A)")  char (str1), char (str2), &
      "events ", char (str3), "..."
  call msg_message ()
  do
```

```
        call simulation%read_event (es_array, .false., complete)
        if (complete)  exit
        if (update_event .or. update_sqme .or. update_weight) then
            call simulation%recalculate (update_sqme, update_weight, &
                recover_beams)
            if (signal_is_pending ())  return
            associate (entry => simulation%entry(simulation%i_prc))
              call entry%update_normalization ()
              call entry%check ()
              call entry%evaluate_expressions ()
              if (signal_is_pending ())  return
              simulation%weight = entry%weight_prc
              call simulation%counter%record (simulation%weight, from_file=.true.)
              call entry%record (simulation%i_mci, from_file=.true.)
            end associate
        else
            associate (entry => simulation%entry(simulation%i_prc))
              call entry%accept_sqme_ref ()
              call entry%accept_weight_ref ()
              call entry%check ()
              call entry%evaluate_expressions ()
              if (signal_is_pending ())  return
              simulation%weight = entry%weight_ref
              call simulation%counter%record (simulation%weight, from_file=.true.)
              call entry%record (simulation%i_mci, from_file=.true.)
            end associate
        end if
        call simulation%calculate_alt_entries ()
        if (signal_is_pending ())  return
        call simulation%write_event (es_array)
      end do
      if (update_sqme .or. update_weight) then
        call simulation%restore_processes ()
      end if
    end subroutine simulation_rescan
```

These routines take care of temporary parameter redefinitions that we want to
take effect while recalculating the matrix elements. We extract the core(s) of
the processes that we are simulating, apply the changes, and make sure that the
changes are actually used. This is the duty of `dispatch_core_update`. When
done, we restore the original versions using `dispatch_core_restore`.

⟨*Simulations: simulation: TBP*⟩+≡
```
    procedure :: update_processes => simulation_update_processes
    procedure :: restore_processes => simulation_restore_processes
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_update_processes (simulation, &
        model, qcd, helicity_selection)
      class(simulation_t), intent(inout) :: simulation
      type(model_t), intent(in), optional, target :: model
      type(qcd_t), intent(in), optional :: qcd
      type(helicity_selection_t), intent(in), optional :: helicity_selection
      integer :: i
      do i = 1, simulation%n_prc
```

```
        call simulation%entry(i)%update_process (model, qcd, helicity_selection)
      end do
    end subroutine simulation_update_processes

    subroutine simulation_restore_processes (simulation)
      class(simulation_t), intent(inout) :: simulation
      integer :: i
      do i = 1, simulation%n_prc
        call simulation%entry(i)%restore_process ()
      end do
    end subroutine simulation_restore_processes
```

22.9.8   Event Stream I/O

Write an event to a generic `eio` event stream. The process index must be selected, or the current index must be available.

⟨*Simulations: simulation: TBP*⟩+≡
```
    generic :: write_event => write_event_eio
    procedure :: write_event_eio => simulation_write_event_eio
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_write_event_eio (object, eio, i_prc)
      class(simulation_t), intent(in) :: object
      class(eio_t), intent(inout) :: eio
      integer, intent(in), optional :: i_prc
      integer :: current
      if (present (i_prc)) then
         current = i_prc
      else
         current = object%i_prc
      end if
      if (current > 0) then
         if (object%split_n_evt > 0) then
            if (object%counter%total > 1 .and. &
                 mod (object%counter%total, object%split_n_evt) == 1) then
               call eio%split_out ()
            end if
         end if
         call eio%output (object%entry(current)%event_t, current)
      else
         call msg_fatal ("Simulation: write event: no process selected")
      end if
    end subroutine simulation_write_event_eio
```

Read an event from a generic `eio` event stream. The event stream element must specify the process within the sample (`i_prc`), the MC group for this process (`i_mci`), the selected term (`i_term`), the selected MC integration `channel`, and the particle set of the event.

   We may encounter EOF, which we indicate by storing 0 for the process index `i_prc`. An I/O error will be reported, and we also abort reading.

⟨*Simulations: simulation: TBP*⟩+≡

```
      generic :: read_event => read_event_eio
      procedure :: read_event_eio => simulation_read_event_eio
```

⟨*Simulations: procedures*⟩+≡
```
      subroutine simulation_read_event_eio (object, eio)
        class(simulation_t), intent(inout) :: object
        class(eio_t), intent(inout) :: eio
        integer :: iostat, current
        call eio%input_i_prc (current, iostat)
        select case (iostat)
        case (0)
           object%i_prc = current
           call eio%input_event (object%entry(current)%event_t, iostat)
        end select
        select case (iostat)
        case (:-1)
           object%i_prc = 0
           object%i_mci = 0
        case (1:)
           call msg_error ("Reading events: I/O error, aborting read")
           object%i_prc = 0
           object%i_mci = 0
        case default
           object%i_mci = object%entry(current)%get_i_mci ()
        end select
      end subroutine simulation_read_event_eio
```

22.9.9   Event Stream Array

Write an event using an array of event I/O streams. The process index must be selected, or the current index must be available.

⟨*Simulations: simulation: TBP*⟩+≡
```
      generic :: write_event => write_event_es_array
      procedure :: write_event_es_array => simulation_write_event_es_array
```

⟨*Simulations: procedures*⟩+≡
```
      subroutine simulation_write_event_es_array (object, es_array)
        class(simulation_t), intent(in) :: object
        class(event_stream_array_t), intent(inout) :: es_array
        integer :: i_prc, event_index
        i_prc = object%i_prc
        if (i_prc > 0) then
           event_index = object%counter%total
           call es_array%output (object%entry(i_prc)%event_t, i_prc, event_index)
        else
           call msg_fatal ("Simulation: write event: no process selected")
        end if
      end subroutine simulation_write_event_es_array
```

Read an event using an array of event I/O streams. Reading is successful if there is an input stream within the array, and if a valid event can be read from that stream. If there is a stream, but EOF is passed when reading the first

item, we switch the channel to output and return failure but no error message, such that new events can be appended to that stream.

⟨*Simulations: simulation: TBP*⟩+≡
```
    generic :: read_event => read_event_es_array
    procedure :: read_event_es_array => simulation_read_event_es_array
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine simulation_read_event_es_array (object, es_array, enable_switch, &
        fail)
      class(simulation_t), intent(inout) :: object
      class(event_stream_array_t), intent(inout) :: es_array
      logical, intent(in) :: enable_switch
      logical, intent(out) :: fail
      integer :: iostat, i_prc
      if (es_array%has_input ()) then
          fail = .false.
          call es_array%input_i_prc (i_prc, iostat)
          select case (iostat)
          case (0)
             object%i_prc = i_prc
             call es_array%input_event (object%entry(i_prc)%event_t, iostat)
          case (:-1)
             write (msg_buffer, "(A,1x,I0,1x,A)")  &
                   "... event file terminates after", &
                   object%counter%read, "events."
             call msg_message ()
             if (enable_switch) then
                call es_array%switch_inout ()
                write (msg_buffer, "(A,1x,I0,1x,A)")  &
                      "Generating remaining ", &
                      object%n_evt_requested - object%counter%read, "events ..."
                call msg_message ()
             end if
             fail = .true.
             return
          end select
          select case (iostat)
          case (0)
             object%i_mci = object%entry(i_prc)%get_i_mci ()
          case default
             write (msg_buffer, "(A,1x,I0,1x,A)")  &
                   "Reading events: I/O error, aborting read after", &
                   object%counter%read, "events."
             call msg_error ()
             object%i_prc = 0
             object%i_mci = 0
             fail = .true.
          end select
      else
          fail = .true.
      end if
    end subroutine simulation_read_event_es_array
```

22.9.10 Recover event

Recalculate the process instance contents, given an event with known particle set. The indices for MC, term, and channel must be already set. The `recalculate` method of the selected entry will import the result into `sqme_prc` and `weight_prc`.

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: recalculate => simulation_recalculate
```

⟨*Simulations: procedures*⟩+≡
```
subroutine simulation_recalculate (simulation, update_sqme, update_weight, &
    recover_beams)
  class(simulation_t), intent(inout) :: simulation
  logical, intent(in) :: update_sqme, update_weight
  logical, intent(in), optional :: recover_beams
  integer :: i_prc
  i_prc = simulation%i_prc
  associate (entry => simulation%entry(i_prc))
    if (update_weight) then
      call simulation%entry(i_prc)%recalculate &
           (update_sqme = update_sqme, recover_beams = recover_beams, &
           weight_factor = entry%get_kinematical_weight ())
    else
      call simulation%entry(i_prc)%recalculate &
           (update_sqme = update_sqme, recover_beams = recover_beams)
    end if
  end associate
end subroutine simulation_recalculate
```

22.9.11 Extract contents

Return an array of pointers to the currently selected processes.

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: get_process_ptr => simulation_get_process_ptr
```

⟨*Simulations: procedures*⟩+≡
```
function simulation_get_process_ptr (simulation) result (ptr)
  class(simulation_t), intent(in) :: simulation
  type(process_ptr_t), dimension(:), allocatable :: ptr
  integer :: i
  allocate (ptr (simulation%n_prc))
  do i = 1, size (ptr)
     ptr(i)%ptr => simulation%entry(i)%get_process_ptr ()
  end do
end function simulation_get_process_ptr
```

Return the MD5 sum that summarizes configuration and integration (but not the event file). Used for initializing the event streams.

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: get_md5sum_prc => simulation_get_md5sum_prc
procedure :: get_md5sum_cfg => simulation_get_md5sum_cfg
procedure :: get_md5sum_alt => simulation_get_md5sum_alt
```

⟨*Simulations: procedures*⟩+≡

```
function simulation_get_md5sum_prc (simulation) result (md5sum)
  class(simulation_t), intent(in) :: simulation
  character(32) :: md5sum
  md5sum = simulation%md5sum_prc
end function simulation_get_md5sum_prc

function simulation_get_md5sum_cfg (simulation) result (md5sum)
  class(simulation_t), intent(in) :: simulation
  character(32) :: md5sum
  md5sum = simulation%md5sum_cfg
end function simulation_get_md5sum_cfg

function simulation_get_md5sum_alt (simulation, i) result (md5sum)
  class(simulation_t), intent(in) :: simulation
  integer, intent(in) :: i
  character(32) :: md5sum
  md5sum = simulation%md5sum_alt(i)
end function simulation_get_md5sum_alt
```

Return data that may be useful for writing event files..

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: get_data => simulation_get_data
```

⟨*Simulations: procedures*⟩+≡

```
function simulation_get_data (simulation, alt) result (data)
  class(simulation_t), intent(in) :: simulation
  logical, intent(in), optional :: alt
  type(event_sample_data_t) :: data
  type(process_t), pointer :: process
  type(beam_data_t), pointer :: beam_data
  integer :: n, i
  logical :: enable_alt
  enable_alt = .true.;  if (present (alt))  enable_alt = alt
  process => simulation%entry(1)%get_process_ptr ()
  beam_data => process%get_beam_data_ptr ()
  if (enable_alt) then
     call data%init (simulation%n_prc, simulation%n_alt)
     do i = 1, simulation%n_alt
        data%md5sum_alt(i) = simulation%get_md5sum_alt (i)
     end do
  else
     call data%init (simulation%n_prc)
  end if
  data%unweighted = simulation%unweighted
  data%negative_weights = simulation%negative_weights
  data%norm_mode = simulation%norm_mode
  n = beam_data_get_n_in (beam_data)
  data%n_beam = n
  data%pdg_beam(:n) = flavor_get_pdg (beam_data_get_flavor (beam_data))
  data%energy_beam(:n) = beam_data_get_energy (beam_data)
  do i = 1, simulation%n_prc
     if (.not. simulation%entry(i)%valid) cycle
     process => simulation%entry(i)%get_process_ptr ()
```

2972

```
        data%proc_num_id(i) = process%get_num_id ()
        if (data%proc_num_id(i) == 0)  data%proc_num_id(i) = i
        if (simulation%entry(i)%has_integral) then
           data%cross_section(i) = simulation%entry(i)%integral
           data%error(i) = simulation%entry(i)%error
        end if
     end do
     data%total_cross_section = sum (data%cross_section)
     data%md5sum_prc = simulation%get_md5sum_prc ()
     data%md5sum_cfg = simulation%get_md5sum_cfg ()
     if (simulation%split_n_evt > 0) then
        data%split_n_evt = simulation%split_n_evt
        data%split_index = simulation%split_index
     end if
  end function simulation_get_data
```

Return a default name for the current event sample. This is the process ID of
the first process.

⟨*Simulations: simulation: TBP*⟩+≡
```
     procedure :: get_default_sample_name => simulation_get_default_sample_name
```

⟨*Simulations: procedures*⟩+≡
```
     function simulation_get_default_sample_name (simulation) result (sample)
       class(simulation_t), intent(in) :: simulation
       type(string_t) :: sample
       type(process_t), pointer :: process
       sample = "whizard"
       if (simulation%n_prc > 0) then
          process => simulation%entry(1)%get_process_ptr ()
          if (associated (process)) then
             sample = process%get_id ()
          end if
       end if
     end function simulation_get_default_sample_name
```

⟨*Simulations: simulation: TBP*⟩+≡
```
     procedure :: is_valid => simulation_is_valid
```

⟨*Simulations: procedures*⟩+≡
```
     function simulation_is_valid (simulation) result (valid)
       class(simulation_t), intent(inout) :: simulation
       logical :: valid
       valid = simulation%valid
     end function simulation_is_valid
```

22.9.12 Auxiliary

Call pacify: eliminate numerical noise.

⟨*Simulations: interfaces*⟩≡
```
     interface pacify
        module procedure pacify_simulation
     end interface
```

⟨*Simulations: procedures*⟩+≡
```
    subroutine pacify_simulation (simulation)
      class(simulation_t), intent(inout) :: simulation
      integer :: i, j
      i = simulation%i_prc
      if (i > 0) then
         call pacify (simulation%entry(i))
         do j = 1, simulation%n_alt
            call pacify (simulation%alt_entry(i,j))
         end do
      end if
    end subroutine pacify_simulation
```

22.9.13   Test

This is the master for calling self-test procedures.
⟨*Simulations: public*⟩+≡
```
    public :: simulations_test
```

⟨*Simulations: tests*⟩≡
```
    subroutine simulations_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Simulations: execute tests*⟩
```
    end subroutine simulations_test
```

Initialization

Initialize a `simulation_t` object, including the embedded event records.
⟨*Simulations: execute tests*⟩≡
```
    call test (simulations_1, "simulations_1", &
         "initialization", &
         u, results)
```
⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_1 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1, procname2
      type(rt_data_t), target :: global
      type(simulation_t), target :: simulation

      write (u, "(A)")  "* Test output: simulations_1"
      write (u, "(A)")  "*   Purpose: initialize simulation"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize processes"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
```

```
      .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_1a"
procname1 = "simulation_1p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("simulations1"), is_known = .true.)
call integrate_process (procname1, global)

libname = "simulation_1b"
procname2 = "sim_extra"

call prepare_test_library (global, libname, 1, [procname2])
call compile_library (libname, global)
call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("simulations2"), is_known = .true.)


write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_string (global%var_list, var_str ("$sample"), &
     var_str ("sim1"), is_known = .true.)
call integrate_process (procname2, global)

call simulation%init ([procname1, procname2], .true., .true., global)
call simulation%init_process_selector ()
call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the first process"
write (u, "(A)")
```

```
      call simulation%write_event (u, i_prc = 1)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_1"

    end subroutine simulations_1
```

Weighted events

Generate events for a single process.

⟨*Simulations: execute tests*⟩+≡
```
      call test (simulations_2, "simulations_2", &
          "weighted events", &
          u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_2 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1
      type(rt_data_t), target :: global
      type(simulation_t), target :: simulation
      type(event_sample_data_t) :: data

      write (u, "(A)")  "* Test output: simulations_2"
      write (u, "(A)")  "*   Purpose: generate events for a single process"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize processes"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("seed"), &
          0, is_known = .true.)

      libname = "simulation_2a"
      procname1 = "simulation_2p"

      call prepare_test_library (global, libname, 1, [procname1])
      call compile_library (libname, global)

      call var_list_append_log (global%var_list, &
          var_str ("?rebuild_events"), .true., intrinsic = .true.)
```

```
      call var_list_set_string (global%var_list, var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$phs_method"), &
           var_str ("single"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$integration_method"),&
           var_str ("midpoint"), is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?vis_history"),&
           .false., is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
           .false., is_known = .true.)

      call var_list_set_real (global%var_list, var_str ("sqrts"),&
           1000._default, is_known = .true.)

      call global%it_list%init ([1], [1000])

      call var_list_set_string (global%var_list, var_str ("$run_id"), &
           var_str ("simulations1"), is_known = .true.)
      call integrate_process (procname1, global)

      write (u, "(A)")  "* Initialize event generation"
      write (u, "(A)")

      call var_list_set_log (global%var_list, var_str ("?unweighted"), &
           .false., is_known = .true.)
      call simulation%init ([procname1], .true., .true., global)
      call simulation%init_process_selector ()

      data = simulation%get_data ()
      call data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Generate three events"
      write (u, "(A)")

      call simulation%generate (3)
      call simulation%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Write the event record for the last event"
      write (u, "(A)")

      call simulation%write_event (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_2"

    end subroutine simulations_2
```

Unweighted events

Generate events for a single process.

⟨*Simulations: execute tests*⟩+≡
```
    call test (simulations_3, "simulations_3", &
        "unweighted events", &
        u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_3 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1
      type(rt_data_t), target :: global
      type(simulation_t), target :: simulation
      type(event_sample_data_t) :: data

      write (u, "(A)")  "* Test output: simulations_3"
      write (u, "(A)")  "*   Purpose: generate unweighted events &
          &for a single process"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize processes"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("seed"), &
          0, is_known = .true.)

      libname = "simulation_3a"
      procname1 = "simulation_3p"

      call prepare_test_library (global, libname, 1, [procname1])
      call compile_library (libname, global)

      call var_list_append_log (global%var_list, &
          var_str ("?rebuild_events"), .true., intrinsic = .true.)

      call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$phs_method"), &
          var_str ("single"), is_known = .true.)
      call var_list_set_string (global%var_list, var_str ("$integration_method"),&
          var_str ("midpoint"), is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?vis_history"),&
          .false., is_known = .true.)
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
          .false., is_known = .true.)
```

```
      call var_list_set_real (global%var_list, var_str ("sqrts"),&
           1000._default, is_known = .true.)

      call global%it_list%init ([1], [1000])

      call var_list_set_string (global%var_list, var_str ("$run_id"), &
           var_str ("simulations1"), is_known = .true.)
      call integrate_process (procname1, global)

      write (u, "(A)")  "* Initialize event generation"
      write (u, "(A)")

      call simulation%init ([procname1], .true., .true., global)
      call simulation%init_process_selector ()

      data = simulation%get_data ()
      call data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Generate three events"
      write (u, "(A)")

      call simulation%generate (3)
      call simulation%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Write the event record for the last event"
      write (u, "(A)")

      call simulation%write_event (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_3"

    end subroutine simulations_3
```

Simulating process with structure functions

Generate events for a single process.

⟨Simulations: execute tests⟩+≡
```
    call test (simulations_4, "simulations_4", &
         "process with structure functions", &
         u, results)
```

⟨Simulations: tests⟩+≡
```
    subroutine simulations_4 (u)
      integer, intent(in) :: u
```

```
type(string_t) :: libname, procname1
type(rt_data_t), target :: global
type(flavor_t) :: flv
type(simulation_t), target :: simulation
type(event_sample_data_t) :: data

write (u, "(A)")  "* Test output: simulations_4"
write (u, "(A)")  "*   Purpose: generate events for a single process &
     &with structure functions"
write (u, "(A)")

write (u, "(A)")  "* Initialize processes"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_4a"
procname1 = "simulation_4p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
     0._default, is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
```

```
    call reset_interaction_counter ()

    call flavor_init (flv, 25, global%model)
    call global%beam_structure%init_sf (flavor_get_name ([flv, flv]), [1])
    call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

    write (u, "(A)")  "* Integrate"
    write (u, "(A)")

    call global%it_list%init ([1], [1000])

    call var_list_set_string (global%var_list, var_str ("$run_id"), &
        var_str ("r1"), is_known = .true.)
    call integrate_process (procname1, global)

    write (u, "(A)")  "* Initialize event generation"
    write (u, "(A)")

    call var_list_set_log (global%var_list, var_str ("?unweighted"), &
        .false., is_known = .true.)
    call var_list_set_string (global%var_list, var_str ("$sample"), &
        var_str ("simulations4"), is_known = .true.)
    call simulation%init ([procname1], .true., .true., global)
    call simulation%init_process_selector ()

    data = simulation%get_data ()
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate three events"
    write (u, "(A)")

    call simulation%generate (3)
    call simulation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write the event record for the last event"
    write (u, "(A)")

    call simulation%write_event (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_4"

end subroutine simulations_4
```

Event I/O

Generate event for a test process, write to file and reread.

⟨*Simulations: execute tests*⟩+≡
```
call test (simulations_5, "simulations_5", &
     "raw event I/O", &
     u, results)
```

⟨*Simulations: tests*⟩+≡
```
subroutine simulations_5 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname1, sample
  type(rt_data_t), target :: global
  type(process_ptr_t) :: process_ptr
  class(eio_t), allocatable :: eio
  type(simulation_t), allocatable, target :: simulation

  write (u, "(A)")  "* Test output: simulations_5"
  write (u, "(A)")  "*   Purpose: generate events for a single process"
  write (u, "(A)")  "*            write to file and reread"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize processes"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()
  call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
       .false., is_known = .true.)
  call var_list_set_int (global%var_list, var_str ("seed"), &
       0, is_known = .true.)

  libname = "simulation_5a"
  procname1 = "simulation_5p"

  call prepare_test_library (global, libname, 1, [procname1])
  call compile_library (libname, global)

  call var_list_append_log (global%var_list, &
       var_str ("?rebuild_events"), .true., intrinsic = .true.)

  call var_list_set_string (global%var_list, var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call var_list_set_string (global%var_list, var_str ("$integration_method"),&
       var_str ("midpoint"), is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?vis_history"),&
       .false., is_known = .true.)
  call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
       .false., is_known = .true.)

  call var_list_set_real (global%var_list, var_str ("sqrts"),&
       1000._default, is_known = .true.)
```

2982

```
call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("simulations5"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations5"
call var_list_set_string (global%var_list, var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

process_ptr%ptr => global%process_stack%get_process_ptr (procname1)

allocate (eio_raw_t :: eio)
call eio%init_out (sample, [process_ptr])

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call simulation%write_event (u)
call simulation%write_event (eio)

call eio%final ()
deallocate (eio)
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()
allocate (eio_raw_t :: eio)
call eio%init_in (sample, [process_ptr])

call simulation%read_event (eio)
call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Recalculate process instance"
```

```
      write (u, "(A)")

      call simulation%recalculate (update_sqme = .true., update_weight = .true.)
      call simulation%entry(simulation%i_prc)%evaluate_expressions ()
      call simulation%write_event (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call eio%final ()
      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_5"

    end subroutine simulations_5
```

Event I/O

Generate event for a real process with structure functions, write to file and reread.

⟨*Simulations: execute tests*⟩+≡
```
    call test (simulations_6, "simulations_6", &
        "raw event I/O with structure functions", &
        u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_6 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1, sample
      type(rt_data_t), target :: global
      type(process_ptr_t) :: process_ptr
      class(eio_t), allocatable :: eio
      type(simulation_t), allocatable, target :: simulation
      type(flavor_t) :: flv

      write (u, "(A)")  "* Test output: simulations_6"
      write (u, "(A)")  "*   Purpose: generate events for a single process"
      write (u, "(A)")  "*            write to file and reread"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and integrate"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("seed"), &
          0, is_known = .true.)
```

2984

```
libname = "simulation_6"
procname1 = "simulation_6p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
     0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)
call global%beam_structure%init_sf (flavor_get_name ([flv, flv]), [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations6"
call var_list_set_string (global%var_list, var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()
```

2985

```
write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

process_ptr%ptr => global%process_stack%get_process_ptr (procname1)

allocate (eio_raw_t :: eio)
call eio%init_out (sample, [process_ptr])

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true., testflag = .true.)
call simulation%write_event (eio)

call eio%final ()
deallocate (eio)
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()
allocate (eio_raw_t :: eio)
call eio%init_in (sample, [process_ptr])

call simulation%read_event (eio)
call simulation%write_event (u, verbose = .true., testflag = .true.)

write (u, "(A)")
write (u, "(A)")  "* Recalculate process instance"
write (u, "(A)")

call simulation%recalculate (update_sqme = .true., update_weight = .true.)
call simulation%entry(simulation%i_prc)%evaluate_expressions ()
call simulation%write_event (u, verbose = .true., testflag = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
call simulation%final ()
call global%final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: simulations_6"
```

```
      end subroutine simulations_6
```

Automatic Event I/O

Generate events with raw-format event file as cache: generate, reread, append.

⟨*Simulations: execute tests*⟩+≡
```
      call test (simulations_7, "simulations_7", &
           "automatic raw event I/O", &
           u, results)
```

⟨*Simulations: tests*⟩+≡
```
      subroutine simulations_7 (u)
        integer, intent(in) :: u
        type(string_t) :: libname, procname1, sample
        type(rt_data_t), target :: global
        type(string_t), dimension(0) :: empty_string_array
        type(event_sample_data_t) :: data
        type(event_stream_array_t) :: es_array
        type(simulation_t), allocatable, target :: simulation
        type(flavor_t) :: flv

        write (u, "(A)")  "* Test output: simulations_7"
        write (u, "(A)")  "*   Purpose: generate events for a single process"
        write (u, "(A)")  "*            write to file and reread"
        write (u, "(A)")

        write (u, "(A)")  "* Initialize process and integrate"
        write (u, "(A)")

        call syntax_model_file_init ()

        call global%global_init ()
        call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
             .false., is_known = .true.)
        call var_list_set_int (global%var_list, var_str ("seed"), &
             0, is_known = .true.)

        libname = "simulation_7"
        procname1 = "simulation_7p"

        call prepare_test_library (global, libname, 1, [procname1])
        call compile_library (libname, global)

        call var_list_append_log (global%var_list, &
             var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
        call var_list_append_log (global%var_list, &
             var_str ("?rebuild_grids"), .true., intrinsic = .true.)
        call var_list_append_log (global%var_list, &
             var_str ("?rebuild_events"), .true., intrinsic = .true.)

        call var_list_set_string (global%var_list, var_str ("$method"), &
             var_str ("unit_test"), is_known = .true.)
```

```
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
     0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)
call global%beam_structure%init_sf (flavor_get_name ([flv, flv]), [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations7"
call var_list_set_string (global%var_list, var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init &
     (sample, [var_str ("raw")], simulation%get_process_ptr (), global, &
     data)

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)
```

```
call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")  "* Re-read the event from file and generate another one"
write (u, "(A)")

call var_list_set_log (global%var_list, &
     var_str ("?rebuild_events"), .false., is_known = .true.)

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init (sample, &
     empty_string_array, simulation%get_process_ptr (), global, data, &
     input = var_str ("raw"))

call simulation%generate (2, es_array)

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)


write (u, "(A)")
write (u, "(A)")  "* Re-read both events from file"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init (sample, &
     empty_string_array, simulation%get_process_ptr (), global, data, &
     input = var_str ("raw"))

call simulation%generate (2, es_array)

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true.)

write (u, "(A)")
```

```
      write (u, "(A)")  "* Cleanup"

      call es_array%final ()
      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_7"

   end subroutine simulations_7
```

Rescanning Events

Generate events and rescan the resulting raw event file.

⟨*Simulations: execute tests*⟩+≡
```
      call test (simulations_8, "simulations_8", &
           "rescan raw event file", &
           u, results)
```
⟨*Simulations: tests*⟩+≡
```
   subroutine simulations_8 (u)
     integer, intent(in) :: u
     type(string_t) :: libname, procname1, sample
     type(rt_data_t), target :: global
     type(string_t), dimension(0) :: empty_string_array
     type(event_sample_data_t) :: data
     type(event_stream_array_t) :: es_array
     type(simulation_t), allocatable, target :: simulation
     type(flavor_t) :: flv

     write (u, "(A)")  "* Test output: simulations_8"
     write (u, "(A)")  "*   Purpose: generate events for a single process"
     write (u, "(A)")  "*            write to file and rescan"
     write (u, "(A)")

     write (u, "(A)")  "* Initialize process and integrate"
     write (u, "(A)")

     call syntax_model_file_init ()

     call global%global_init ()
     call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
          .false., is_known = .true.)
     call var_list_set_int (global%var_list, var_str ("seed"), &
          0, is_known = .true.)

     libname = "simulation_8"
     procname1 = "simulation_8p"

     call prepare_test_library (global, libname, 1, [procname1])
     call compile_library (libname, global)

     call var_list_append_log (global%var_list, &
```

```
      var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
     0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)
call global%beam_structure%init_sf (flavor_get_name ([flv, flv]), [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations8"
call var_list_set_string (global%var_list, var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
```

```
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init &
     (sample, [var_str ("raw")], simulation%get_process_ptr (), global, &
     data)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true., testflag = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], .false., .false., global)
call simulation%init_process_selector ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)  = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, &
     empty_string_array, simulation%get_process_ptr (), global, data, &
     input = var_str ("raw"), input_sample = sample, allow_switch = .false.)

call simulation%rescan (1, es_array, &
     update_event = .false., &
     update_sqme = .false., &
     update_weight = .false., &
     recover_beams = .false., &
     global = global)

write (u, "(A)")

call pacify (simulation%entry(simulation%i_prc))
call simulation%write_event (u, verbose = .true., testflag = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read again and recalculate"
```

```
      write (u, "(A)")

      call reset_interaction_counter ()

      allocate (simulation)
      call simulation%init ([procname1], .false., .false., global)
      call simulation%init_process_selector ()

      data%md5sum_prc = simulation%get_md5sum_prc ()
      data%md5sum_cfg = ""
      write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
      write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
      call es_array%init (sample, &
           empty_string_array, simulation%get_process_ptr (), global, data, &
           input = var_str ("raw"), input_sample = sample, allow_switch = .false.)

      call simulation%rescan (1, es_array, &
           update_event = .true., &
           update_sqme = .true., &
           update_weight = .false., &
           recover_beams = .false., &
           global = global)

      write (u, "(A)")

      call pacify (simulation%entry(simulation%i_prc))
      call simulation%write_event (u, verbose = .true., testflag = .true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call es_array%final ()
      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_8"

    end subroutine simulations_8
```

Rescanning Check

Generate events and rescan with process mismatch.

⟨*Simulations: execute tests*⟩+≡
```
    call test (simulations_9, "simulations_9", &
         "rescan mismatch", &
         u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_9 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1, sample
      type(rt_data_t), target :: global
```

```
type(string_t), dimension(0) :: empty_string_array
type(event_sample_data_t) :: data
type(event_stream_array_t) :: es_array
type(simulation_t), allocatable, target :: simulation
type(flavor_t) :: flv
logical :: error

write (u, "(A)")  "* Test output: simulations_9"
write (u, "(A)")  "*   Purpose: generate events for a single process"
write (u, "(A)")  "*              write to file and rescan"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and integrate"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_9"
procname1 = "simulation_9p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
     0._default, is_known = .true.)
```

```
call flavor_init (flv, 25, global%model)
call global%beam_structure%init_sf (flavor_get_name ([flv, flv]), [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations9"
call var_list_set_string (global%var_list, var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init &
     (sample, [var_str ("raw")], simulation%get_process_ptr (), global, &
     data)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")  "* Initialize event generation for different parameters"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
```

```
      call simulation%init ([procname1, procname1], .false., .false., global)
      call simulation%init_process_selector ()

      call simulation%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Attempt to re-read the events (should fail)"
      write (u, "(A)")

      data%md5sum_prc = simulation%get_md5sum_prc ()
      data%md5sum_cfg = ""
      write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
      write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
      call es_array%init (sample, &
           empty_string_array, simulation%get_process_ptr (), global, data, &
           input = var_str ("raw"), input_sample = sample, &
           allow_switch = .false., error = error)

      write (u, "(1x,A,L1)")  "error = ", error

      call simulation%rescan (1, es_array, &
           update_event = .false., &
           update_sqme = .false., &
           update_weight = .false., &
           recover_beams = .false., &
           global = global)

      call es_array%final ()
      call simulation%final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_9"

    end subroutine simulations_9
```

Alternative weights

Generate an event for a single process and reweight it in a simultaneous calculation.

⟨*Simulations: execute tests*⟩+≡
```
    call test (simulations_10, "simulations_10", &
         "alternative weight", &
         u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_10 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1, expr_text
      type(rt_data_t), target :: global
      type(rt_data_t), dimension(1), target :: alt_env
      type(ifile_t) :: ifile
      type(stream_t) :: stream
```

```
type(parse_tree_t) :: pt_weight
type(simulation_t), target :: simulation
type(event_sample_data_t) :: data

write (u, "(A)")  "* Test output: simulations_10"
write (u, "(A)")  "*   Purpose: reweight event"
write (u, "(A)")

write (u, "(A)")  "* Initialize processes"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_pexpr_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_10a"
procname1 = "simulation_10p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("simulations1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize alternative environment with custom weight"
```

2997

```
write (u, "(A)")

call alt_env(1)%local_init (global)
call alt_env(1)%link (global)

expr_text = "2"
write (u, "(A,A)")  "weight = ", char (expr_text)
write (u, *)

call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)
alt_env(1)%pn%weight_expr => parse_tree_get_root_ptr (pt_weight)
call alt_env(1)%write_expr (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
call simulation%init ([procname1], .true., .true., global, alt_env)
call simulation%init_process_selector ()

data = simulation%get_data ()
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the last event"
write (u, "(A)")

call simulation%write_event (u)

write (u, "(A)")
write (u, "(A)")  "* Write the event record for the alternative setup"
write (u, "(A)")

call simulation%write_alt_event (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call simulation%final ()
call global%final ()
```

```
      call syntax_model_file_final ()
      call syntax_pexpr_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_10"

    end subroutine simulations_10
```

Decays

Generate an event with subsequent partonic decays.

⟨*Simulations: execute tests*⟩+≡
```
      call test (simulations_11, "simulations_11", &
           "decay", &
           u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_11 (u)
      integer, intent(in) :: u
      type(rt_data_t), target :: global
      type(prclib_entry_t), pointer :: lib
      type(string_t) :: prefix, procname1, procname2
      type(simulation_t), target :: simulation

      write (u, "(A)")  "* Test output: simulations_11"
      write (u, "(A)")  "*   Purpose: apply decay"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize processes"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      allocate (lib)
      call global%add_prclib (lib)

      call var_list_set_int (global%var_list, var_str ("seed"), &
           0, is_known = .true.)

      prefix = "simulation_11"
      procname1 = prefix // "_p"
      procname2 = prefix // "_d"
      call prepare_testbed &
           (global%prclib, global%process_stack, global%model_list, &
           global%model, prefix, global%os_data, &
           scattering=.true., decay=.true.)
      call model_set_unstable (global%model, 25, [procname2])

      write (u, "(A)")  "* Initialize simulation object"
      write (u, "(A)")

      call simulation%init ([procname1], .true., .true., global)
```

2999

```
      call simulation%init_process_selector ()

      write (u, "(A)")  "* Generate event"
      write (u, "(A)")

      call simulation%generate (1)
      call simulation%write (u)

      write (u, *)

      call simulation%write_event (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"
      write (u, "(A)")

      call simulation%final ()
      call global%final ()

      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_11"

    end subroutine simulations_11
```

Split Event Files

Generate event for a real process with structure functions and write to file, accepting a limit for the number of events per file.

⟨*Simulations: execute tests*⟩+≡
```
    call test (simulations_12, "simulations_12", &
         "split event files", &
         u, results)
```

⟨*Simulations: tests*⟩+≡
```
    subroutine simulations_12 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1, sample
      type(rt_data_t), target :: global
      type(process_ptr_t) :: process_ptr
      class(eio_t), allocatable :: eio
      type(simulation_t), allocatable, target :: simulation
      type(flavor_t) :: flv
      integer :: i_evt

      write (u, "(A)")  "* Test output: simulations_12"
      write (u, "(A)")  "*   Purpose: generate events for a single process"
      write (u, "(A)")  "*            and write to split event files"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and integrate"
      write (u, "(A)")
```

```
call syntax_model_file_init ()

call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_12"
procname1 = "simulation_12p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call var_list_append_log (global%var_list, &
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call var_list_append_log (global%var_list, &
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known = .true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)

call var_list_set_real (global%var_list, var_str ("sqrts"),&
     1000._default, is_known = .true.)
call var_list_set_real (global%var_list, var_str ("ms"), &
     0._default, is_known = .true.)

call flavor_init (flv, 25, global%model)

call global%it_list%init ([1], [1000])

call var_list_set_string (global%var_list, var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call var_list_set_log (global%var_list, var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations_12"
call var_list_set_string (global%var_list, var_str ("$sample"), &
     sample, is_known = .true.)
```

```
      call var_list_set_int (global%var_list, var_str ("sample_split_n_evt"), &
           2, is_known = .true.)
      call var_list_set_int (global%var_list, var_str ("sample_split_index"), &
           42, is_known = .true.)
      allocate (simulation)
      call simulation%init ([procname1], .true., .true., global)
      call simulation%init_process_selector ()

      call simulation%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize ASCII event file"
      write (u, "(A)")

      process_ptr%ptr => global%process_stack%get_process_ptr (procname1)

      allocate (eio_ascii_short_t :: eio)
      select type (eio)
      class is (eio_ascii_t);  call eio%set_parameters ()
      end select
      call eio%init_out (sample, [process_ptr], data = simulation%get_data ())

      write (u, "(A)")  "* Generate 5 events, distributed among three files"

      do i_evt = 1, 5
         call simulation%generate (1)
         call simulation%write_event (eio)
      end do

      call eio%final ()
      deallocate (eio)
      call simulation%final ()
      deallocate (simulation)

      write (u, *)
      call display_file ("simulations_12.42.short.evt", u)
      write (u, *)
      call display_file ("simulations_12.43.short.evt", u)
      write (u, *)
      call display_file ("simulations_12.44.short.evt", u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_12"

    end subroutine simulations_12
```

Auxiliary: display file contents.

⟨*Simulations: tests*⟩+≡

```
    subroutine display_file (file, u)
      character(*), intent(in) :: file
      integer, intent(in) :: u
      character(256) :: buffer
      integer :: u_file
      write (u, "(3A)")  "* Contents of file '", file, "':"
      write (u, *)
      u_file = free_unit ()
      open (u_file, file = file, action = "read", status = "old")
      do
         read (u_file, "(A)", end = 1)  buffer
         write (u, "(A)")  trim (buffer)
      end do
1   continue
    end subroutine display_file
```

Chapter 23

Top level API

23.1 Commands

This module defines the command language of the main input file.

⟨commands.f90⟩≡
  ⟨*File header*⟩

  module commands

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩
    use constants !NODEP!
  ⟨*Use file utils*⟩
    use limits, only: FMT_14, FMT_19, SHOW_BUFFER_SIZE !NODEP!
    use limits, only: DEFAULT_ANALYSIS_FILENAME !NODEP!
    use limits, only: FORBIDDEN_ENDINGS1 !NODEP!
    use limits, only: FORBIDDEN_ENDINGS2 !NODEP!
    use limits, only: FORBIDDEN_ENDINGS3 !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use sm_qcd
    use pdf_builtin !NODEP!
    use sorting
    use sf_lhapdf
    use os_interface
    use ifiles
    use lexers
    use syntax_rules
    use parser
    use analysis
    use pdg_arrays
    use variables
    use expressions
    use models
    use interactions
    use flavors
    use polarizations
    use beams

```
      use particle_specifiers
      use process_libraries
      use processes
      use prclib_stacks
      use slha_interface
      use user_files
      use eio_data
      use rt_data
      use dispatch
      use process_configurations
      use compilations
      use integrations
      use event_streams
      use simulations
```

⟨*Standard module head*⟩

⟨*Commands: public*⟩

⟨*Commands: types*⟩

⟨*Commands: variables*⟩

⟨*Commands: parameters*⟩

⟨*Commands: interfaces*⟩

```
contains
```

⟨*Commands: procedures*⟩

⟨*Commands: tests*⟩

```
end module commands
```

23.1.1   The command type

The command type is a generic type that holds any command, compiled for execution.

Each command may come with its own local environment. The command list that determines this environment is allocated as `options`, if necessary. (It has to be allocated as a pointer because the type definition is recursive.) The local environment is available as a pointer which either points to the global environment, or is explicitly allocated and initialized.

⟨*Commands: types*⟩≡
```
    type, abstract :: command_t
      type(parse_node_t), pointer :: pn => null ()
      class(command_t), pointer :: next => null ()
      type(parse_node_t), pointer :: pn_opt => null ()
      type(command_list_t), pointer :: options => null ()
      type(rt_data_t), pointer :: local => null ()
    contains
    ⟨Commands: command: TBP⟩
```

```
        end type command_t
```

Finalizer: If there is an option list, finalize the option list and deallocate. If not, the local environment is just a pointer.

⟨*Commands: command: TBP*⟩≡
```
    procedure :: final => command_final
```

⟨*Commands: procedures*⟩≡
```
    recursive subroutine command_final (cmd)
      class(command_t), intent(inout) :: cmd
      if (associated (cmd%options)) then
         call cmd%options%final ()
         deallocate (cmd%options)
         call cmd%local%local_final ()
         deallocate (cmd%local)
      else
         cmd%local => null ()
      end if
    end subroutine command_final
```

Allocate a command with the appropriate concrete type. Store the parse node pointer in the command object, so we can reference to it when compiling.

⟨*Commands: procedures*⟩+≡
```
    subroutine dispatch_command (command, pn)
      class(command_t), intent(inout), pointer :: command
      type(parse_node_t), intent(in), target :: pn
      select case (char (parse_node_get_rule_key (pn)))
      case ("cmd_model")
         allocate (cmd_model_t :: command)
      case ("cmd_library")
         allocate (cmd_library_t :: command)
      case ("cmd_process")
         allocate (cmd_process_t :: command)
      case ("cmd_compile")
         allocate (cmd_compile_t :: command)
      case ("cmd_exec")
         allocate (cmd_exec_t :: command)
       case ("cmd_num", "cmd_complex", "cmd_real", "cmd_int", &
             "cmd_log_decl", "cmd_log", "cmd_string", "cmd_string_decl", &
             "cmd_alias", "cmd_result")
         allocate (cmd_var_t :: command)
      case ("cmd_slha")
         allocate (cmd_slha_t :: command)
      case ("cmd_show")
         allocate (cmd_show_t :: command)
      case ("cmd_clear")
         allocate (cmd_clear_t :: command)
      case ("cmd_expect")
         allocate (cmd_expect_t :: command)
      case ("cmd_beams")
         allocate (cmd_beams_t :: command)
      case ("cmd_beams_pol_density")
         allocate (cmd_beams_pol_density_t :: command)
```

```
case ("cmd_beams_pol_fraction")
   allocate (cmd_beams_pol_fraction_t :: command)
case ("cmd_beams_momentum")
   allocate (cmd_beams_momentum_t :: command)
case ("cmd_beams_theta")
   allocate (cmd_beams_theta_t :: command)
case ("cmd_beams_phi")
   allocate (cmd_beams_phi_t :: command)
case ("cmd_cuts")
   allocate (cmd_cuts_t :: command)
case ("cmd_scale")
   allocate (cmd_scale_t :: command)
case ("cmd_fac_scale")
   allocate (cmd_fac_scale_t :: command)
case ("cmd_ren_scale")
   allocate (cmd_ren_scale_t :: command)
case ("cmd_weight")
   allocate (cmd_weight_t :: command)
case ("cmd_selection")
   allocate (cmd_selection_t :: command)
case ("cmd_reweight")
   allocate (cmd_reweight_t :: command)
case ("cmd_iterations")
   allocate (cmd_iterations_t :: command)
case ("cmd_integrate")
   allocate (cmd_integrate_t :: command)
case ("cmd_observable")
   allocate (cmd_observable_t :: command)
case ("cmd_histogram")
   allocate (cmd_histogram_t :: command)
case ("cmd_plot")
   allocate (cmd_plot_t :: command)
case ("cmd_graph")
   allocate (cmd_graph_t :: command)
case ("cmd_record")
   allocate (cmd_record_t :: command)
case ("cmd_analysis")
   allocate (cmd_analysis_t :: command)
case ("cmd_alt_setup")
   allocate (cmd_alt_setup_t :: command)
case ("cmd_unstable")
   allocate (cmd_unstable_t :: command)
case ("cmd_stable")
   allocate (cmd_stable_t :: command)
case ("cmd_polarized")
   allocate (cmd_polarized_t :: command)
case ("cmd_unpolarized")
   allocate (cmd_unpolarized_t :: command)
case ("cmd_sample_format")
   allocate (cmd_sample_format_t :: command)
case ("cmd_simulate")
   allocate (cmd_simulate_t :: command)
case ("cmd_rescan")
   allocate (cmd_rescan_t :: command)
```

```
      case ("cmd_write_analysis")
         allocate (cmd_write_analysis_t :: command)
      case ("cmd_compile_analysis")
         allocate (cmd_compile_analysis_t :: command)
      case ("cmd_open_out")
         allocate (cmd_open_out_t :: command)
      case ("cmd_close_out")
         allocate (cmd_close_out_t :: command)
      case ("cmd_printf")
         allocate (cmd_printf_t :: command)
      case ("cmd_scan")
         allocate (cmd_scan_t :: command)
      case ("cmd_if")
         allocate (cmd_if_t :: command)
      case ("cmd_include")
         allocate (cmd_include_t :: command)
      case ("cmd_quit")
         allocate (cmd_quit_t :: command)
      case default
         print *, char (parse_node_get_rule_key (pn))
         call msg_bug ("Command not implemented")
      end select
      command%pn => pn
    end subroutine dispatch_command
```

Output. We allow for indentation so we can display a command tree.

⟨*Commands: command: TBP*⟩+≡
```
      procedure (command_write), deferred :: write
```

⟨*Commands: interfaces*⟩≡
```
      abstract interface
        subroutine command_write (cmd, unit, indent)
          import
          class(command_t), intent(in) :: cmd
          integer, intent(in), optional :: unit, indent
        end subroutine command_write
      end interface
```

Compile a command. The command type is already fixed, so this is a deferred type-bound procedure.

⟨*Commands: command: TBP*⟩+≡
```
      procedure (command_compile), deferred :: compile
```

⟨*Commands: interfaces*⟩+≡
```
      abstract interface
        subroutine command_compile (cmd, global)
          import
          class(command_t), intent(inout) :: cmd
          type(rt_data_t), intent(inout), target :: global
        end subroutine command_compile
      end interface
```

Execute a command. This will use and/or modify the runtime data set. If the
`quit` flag is set, the caller should terminate command execution.

⟨*Commands: command: TBP*⟩+≡
```
    procedure (command_execute), deferred :: execute
```

⟨*Commands: interfaces*⟩+≡
```
    abstract interface
       subroutine command_execute (cmd, global)
         import
         class(command_t), intent(inout) :: cmd
         type(rt_data_t), intent(inout), target :: global
       end subroutine command_execute
    end interface
```

23.1.2   Options

The `options` command list is allocated, initialized, and executed, if the com-
mand is associated with an option text in curly braces. If present, a separate
local runtime data set `local` will be allocated and initialized; otherwise, `local`
becomes a pointer to the global dataset.

For output, we indent the options list.

⟨*Commands: command: TBP*⟩+≡
```
    procedure :: write_options => command_write_options
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_write_options (cmd, unit, indent)
      class(command_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: ind
      ind = 1;  if (present (indent))  ind = indent + 1
      if (associated (cmd%options))  call cmd%options%write (unit, ind)
    end subroutine command_write_options
```

Compile the options list, if any. This implies initialization of the local environ-
ment. Should be done once the `pn_opt` node has been assigned (if applicable),
but before the actual command compilation.

⟨*Commands: command: TBP*⟩+≡
```
    procedure :: compile_options => command_compile_options
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_compile_options (cmd, global)
      class(command_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      if (associated (cmd%pn_opt)) then
         allocate (cmd%local)
         call cmd%local%local_init (global)
         call global%copy_globals (cmd%local)
         allocate (cmd%options)
         call cmd%options%compile (cmd%pn_opt, cmd%local)
         call global%restore_globals (cmd%local)
      else
         cmd%local => global
```

```
      end if
   end subroutine command_compile_options
```

Execute options. First prepare the local environment, then execute the command list.

⟨*Commands: command: TBP*⟩+≡
```
   procedure :: execute_options => cmd_execute_options
```

⟨*Commands: procedures*⟩+≡
```
   recursive subroutine cmd_execute_options (cmd, global)
     class(command_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     if (associated (cmd%options)) then
        call cmd%local%link (global)
        call cmd%options%execute (cmd%local)
     end if
   end subroutine cmd_execute_options
```

This must be called after the parent command has been executed, to undo temporary modifications to the global environment.

⟨*Commands: command: TBP*⟩+≡
```
   procedure :: reset_options => cmd_reset_options
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_reset_options (cmd, global)
     class(command_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     if (associated (cmd%options)) then
        call global%restore (cmd%local)
     end if
   end subroutine cmd_reset_options
```

23.1.3 Specific command types

Model configuration

The command declares a model, looks for the specified file and loads it.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_model_t
     private
     type(string_t) :: name
    contains
    ⟨Commands: cmd model: TBP⟩
   end type cmd_model_t
```

Output

⟨*Commands: cmd model: TBP*⟩≡
```
   procedure :: write => cmd_model_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_model_write (cmd, unit, indent)
  class(cmd_model_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,'""',A,'""')")  "model =", char (cmd%name)
end subroutine cmd_model_write
```

Compile. Get the model name and read the model from file, so it is readily available when the command list is executed.

⟨*Commands: cmd model: TBP*⟩+≡

```
procedure :: compile => cmd_model_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_model_compile (cmd, global)
  class(cmd_model_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_name
  type(string_t) :: filename
  pn_name => parse_node_get_sub_ptr (cmd%pn, 3)
  cmd%name = parse_node_get_string (pn_name)
  filename = cmd%name // ".mdl"
  call global%read_model (cmd%name, filename)
end subroutine cmd_model_compile
```

Execute: Insert a pointer into the global data record and reassign the variable list.

⟨*Commands: cmd model: TBP*⟩+≡

```
procedure :: execute => cmd_model_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_model_execute (cmd, global)
  class(cmd_model_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  logical :: new_model
  new_model = .true.
  if (associated (global%model)) then
     new_model = global%model%get_name () /= cmd%name
  end if
  if (new_model) then
     if (global%model_list%model_exists (cmd%name)) then
        call msg_message ("Switching to model '" &
             // char (cmd%name) &
             // "': reassigning model parameters")
        call global%select_model (cmd%name)
     else
        call msg_fatal ("Switching to model '" &
             // char (cmd%name) &
             // "': unknown model name")
     end if
  end if
end subroutine cmd_model_execute
```

3011

Library configuration

We configure a process library that should hold the subsequently defined processes. If the referenced library exists already, just make it the currently active one.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_library_t
      private
      type(string_t) :: name
    contains
    ⟨Commands: cmd library: TBP⟩
    end type cmd_library_t
```

Output.

⟨*Commands: cmd library: TBP*⟩≡
```
    procedure :: write => cmd_library_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_library_write (cmd, unit, indent)
      class(cmd_library_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit)
      call write_indent (u, indent)
      write (u, "(1x,A,1x,'""',A,'""')")  "library =", char (cmd%name)
    end subroutine cmd_library_write
```

Compile. Get the library name.

⟨*Commands: cmd library: TBP*⟩+≡
```
    procedure :: compile => cmd_library_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_library_compile (cmd, global)
      class(cmd_library_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_name
      pn_name => parse_node_get_sub_ptr (cmd%pn, 3)
      cmd%name = parse_node_get_string (pn_name)
    end subroutine cmd_library_compile
```

Execute: Initialize a new library and push it on the library stack (if it does not yet exist). Insert a pointer to the library into the global data record. Then, try to load the library unless the `rebuild` flag is set.

⟨*Commands: cmd library: TBP*⟩+≡
```
    procedure :: execute => cmd_library_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_library_execute (cmd, global)
      class(cmd_library_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(prclib_entry_t), pointer :: lib_entry
```

```
      type(process_library_t), pointer :: lib
      logical :: rebuild_library
      lib => global%prclib_stack%get_library_ptr (cmd%name)
      rebuild_library = &
           var_list_get_lval (global%var_list, var_str ("?rebuild_library"))
      if (.not. (associated (lib))) then
         allocate (lib_entry)
         call lib_entry%init (cmd%name)
         lib => lib_entry%process_library_t
         call global%add_prclib (lib_entry)
      else
         call global%update_prclib (lib)
      end if
      if (associated (lib) .and. .not. rebuild_library) then
         call lib%update_status (global%os_data)
      end if
    end subroutine cmd_library_execute
```

Process configuration

We define a process-configuration command as a specific type. The incoming
and outgoing particles are given evaluation-trees which we transform to PDG-
code arrays. For transferring to O'MEGA, they are reconverted to strings.

   For the incoming particles, we store parse nodes individually. We do not yet
resolve the outgoing state, so we store just a single parse node.

   This also includes the choice of method for the corresponding process: omega
for O'MEGA matrix elements, test for special processes generated by WHIZARD,
external for using an external matrix element and dipole for generating dipole
subtraction terms.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_process_t
      private
      type(string_t) :: id
      integer :: n_in  = 0
      type(parse_node_p), dimension(:), allocatable :: pn_pdg_in
      type(parse_node_t), pointer :: pn_out => null ()
    contains
    ⟨Commands: cmd process: TBP⟩
    end type cmd_process_t
```

Output. The particle expressions are not resolved, so we just list the number of
incoming particles.

⟨*Commands: cmd process: TBP*⟩≡
```
    procedure :: write => cmd_process_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_process_write (cmd, unit, indent)
      class(cmd_process_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
```

```
      write (u, "(1x,A,A,A,I0,A)")  "process: ", char (cmd%id), " (", &
           size (cmd%pn_pdg_in), " -> X)"
      call cmd%write_options (u, indent)
   end subroutine cmd_process_write
```

Compile. Find and assign the parse nodes.

⟨*Commands: cmd process: TBP*⟩+≡
```
    procedure :: compile => cmd_process_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_process_compile (cmd, global)
      class(cmd_process_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_id, pn_in, pn_codes
      integer :: i
      pn_id => parse_node_get_sub_ptr (cmd%pn, 2)
      pn_in  => parse_node_get_next_ptr (pn_id, 2)
      cmd%pn_out => parse_node_get_next_ptr (pn_in, 2)
      cmd%pn_opt => parse_node_get_next_ptr (cmd%pn_out)
      call cmd%compile_options (global)
      cmd%id = parse_node_get_string (pn_id)
      cmd%n_in  = parse_node_get_n_sub (pn_in)
      pn_codes => parse_node_get_sub_ptr (pn_in)
      allocate (cmd%pn_pdg_in (cmd%n_in))
      do i = 1, cmd%n_in
         cmd%pn_pdg_in(i)%ptr => pn_codes
         pn_codes => parse_node_get_next_ptr (pn_codes)
      end do
    end subroutine cmd_process_compile
```

Command execution. Evaluate the subevents, transform PDG codes into strings, and add the current process configuration to the process library.

The initial state will be unique (one or two particles). For the final state, we allow for expressions. The expressions will be expanded until we have a sum of final states. Each distinct final state will get its own process component.

To identify equivalent final states, we transform the final state into an array of PDG codes, which we sort and compare. If a particle entry is actually a PDG array, only the first entry in the array is used for the comparison. The user should make sure that there is no overlap between different particles or arrays which would make the expansion ambiguous.

⟨*Commands: cmd process: TBP*⟩+≡
```
    procedure :: execute => cmd_process_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_process_execute (cmd, global)
      class(cmd_process_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(pdg_array_t) :: pdg_in, pdg_out
      type(pdg_array_t), dimension(:), allocatable :: pdg_out_tab
      type(string_t), dimension(:), allocatable :: prt_in
      type(string_t) :: prt_out, prt_out1
      type(process_configuration_t) :: prc_config
      type(prt_expr_t) :: prt_expr_out
```

```
      type(prt_spec_t), dimension(:), allocatable :: prt_spec_in
      type(prt_spec_t), dimension(:), allocatable :: prt_spec_out
      integer, dimension(:), allocatable :: pdg
      integer, dimension(:), allocatable :: i_term
      integer :: i, j, n_in, n_out, n_terms, n_components

      n_in = size (cmd%pn_pdg_in)
      allocate (prt_in (n_in), prt_spec_in (n_in))
      do i = 1, n_in
         pdg_in = &
              eval_pdg_array (cmd%pn_pdg_in(i)%ptr, cmd%local%var_list)
         prt_in(i) = make_flavor_string (pdg_in, cmd%local%model)
         prt_spec_in(i) = new_prt_spec (prt_in(i))
      end do
      call compile_prt_expr &
           (prt_expr_out, cmd%pn_out, cmd%local%var_list, cmd%local%model)
      call prt_expr_out%expand ()
      n_terms = prt_expr_out%get_n_terms ()
      allocate (pdg_out_tab (n_terms))
      allocate (i_term (n_terms), source = 0)
      n_components = 0
      SCAN_COMPONENTS: do i = 1, n_terms
         if (allocated (pdg))  deallocate (pdg)
         call prt_expr_out%term_to_array (prt_spec_out, i)
         n_out = size (prt_spec_out)
         allocate (pdg (n_out))
         do j = 1, n_out
            prt_out = prt_spec_out(j)%to_string ()
            call split (prt_out, prt_out1, ":")
            pdg(j) = model_get_particle_pdg (cmd%local%model, prt_out1)
         end do
         pdg_out = sort (pdg)
         do j = 1, n_components
            if (pdg_out == pdg_out_tab(j))  cycle SCAN_COMPONENTS
         end do
         n_components = n_components + 1
         i_term(n_components) = i
         pdg_out_tab(n_components) = pdg_out
      end do SCAN_COMPONENTS
      call prc_config%init (cmd%id, n_in, n_components, cmd%local)
      do i = 1, n_components
         call prt_expr_out%term_to_array (prt_spec_out, i_term(i))
         call prc_config%setup_component (i, prt_spec_in, prt_spec_out, cmd%local)
      end do
      call prc_config%record (cmd%local)

    end subroutine cmd_process_execute
```

This is a method of the eval tree, but cannot be coded inside the `expressions` module since it uses the `model` and `flv` types which are not available there.

⟨*Commands: procedures*⟩+≡
```
    function make_flavor_string (aval, model) result (prt)
      type(string_t) :: prt
```

```
      type(pdg_array_t), intent(in) :: aval
      type(model_t), intent(in), target :: model
      integer, dimension(:), allocatable :: pdg
      type(flavor_t), dimension(:), allocatable :: flv
      integer :: i
      pdg = aval
      allocate (flv (size (pdg)))
      call flavor_init (flv, pdg, model)
      if (size (pdg) /= 0) then
         prt = flavor_get_name (flv(1))
         do i = 2, size (flv)
            prt = prt // ":" // flavor_get_name (flv(i))
         end do
      else
         prt = "?"
      end if
   end function make_flavor_string
```

Compile a (possible nested) expression, to obtain a particle-specifier expression
which we can process further.

⟨*Commands: procedures*⟩+≡

```
    recursive subroutine compile_prt_expr (prt_expr, pn, var_list, model)
      type(prt_expr_t), intent(out) :: prt_expr
      type(parse_node_t), intent(in), target :: pn
      type(var_list_t), intent(in), target :: var_list
      type(model_t), intent(in), target :: model
      type(parse_node_t), pointer :: pn_entry, pn_term, pn_addition
      type(pdg_array_t) :: pdg
      type(string_t) :: prt_string
      integer :: n_entry, n_term, i
      select case (char (parse_node_get_rule_key (pn)))
      case ("prt_state_list")
         n_entry = parse_node_get_n_sub (pn)
         pn_entry => parse_node_get_sub_ptr (pn)
         if (n_entry == 1) then
            call compile_prt_expr (prt_expr, pn_entry, var_list, model)
         else
            call prt_expr%init_list (n_entry)
            select type (x => prt_expr%x)
            type is (prt_spec_list_t)
               do i = 1, n_entry
                  call compile_prt_expr (x%expr(i), pn_entry, var_list, model)
                  pn_entry => parse_node_get_next_ptr (pn_entry)
               end do
            end select
         end if
      case ("prt_state_sum")
         n_term = parse_node_get_n_sub (pn)
         pn_term => parse_node_get_sub_ptr (pn)
         pn_addition => pn_term
         if (n_term == 1) then
            call compile_prt_expr (prt_expr, pn_term, var_list, model)
         else
```

```
                  call prt_expr%init_sum (n_term)
                  select type (x => prt_expr%x)
                  type is (prt_spec_sum_t)
                     do i = 1, n_term
                        call compile_prt_expr (x%expr(i), pn_term, var_list, model)
                        pn_addition => parse_node_get_next_ptr (pn_addition)
                        if (associated (pn_addition)) &
                             pn_term => parse_node_get_sub_ptr (pn_addition, 2)
                     end do
                  end select
               end if
         case ("cexpr")
            pdg = eval_pdg_array (pn, var_list)
            prt_string = make_flavor_string (pdg, model)
            call prt_expr%init_spec (new_prt_spec (prt_string))
         case default
            call parse_node_write_rec (pn)
            call msg_bug ("compile prt expr: impossible syntax rule")
         end select
    end subroutine compile_prt_expr
```

Process compilation

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_compile_t
      private
      type(string_t), dimension(:), allocatable :: libname
      logical :: make_executable = .false.
      type(string_t) :: exec_name
     contains
    ⟨Commands: cmd compile: TBP⟩
    end type cmd_compile_t
```

Output: list all libraries to be compiled.
⟨*Commands: cmd compile: TBP*⟩≡
```
    procedure :: write => cmd_compile_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_write (cmd, unit, indent)
      class(cmd_compile_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "compile ("
      if (allocated (cmd%libname)) then
         do i = 1, size (cmd%libname)
            if (i > 1)  write (u, "(A,1x)", advance="no")  ","
            write (u, "('""',A,'""')", advance="no")  char (cmd%libname(i))
         end do
      end if
      write (u, "(A)")  ")"
    end subroutine cmd_compile_write
```

Compile the libraries specified in the argument. If the argument is empty, compile all libraries which can be found in the process library stack.

⟨*Commands: cmd compile: TBP*⟩+≡

```
procedure :: compile => cmd_compile_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_compile_compile (cmd, global)
  class(cmd_compile_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_arg, pn_lib
  type(parse_node_t), pointer :: pn_exec_name_spec, pn_exec_name
  integer :: n_lib, i
  pn_cmd => parse_node_get_sub_ptr (cmd%pn)
  pn_clause => parse_node_get_sub_ptr (pn_cmd)
  pn_exec_name_spec => parse_node_get_sub_ptr (pn_clause, 2)
  if (associated (pn_exec_name_spec)) then
     pn_exec_name => parse_node_get_sub_ptr (pn_exec_name_spec, 2)
  else
     pn_exec_name => null ()
  end if
  pn_arg => parse_node_get_next_ptr (pn_clause)
  cmd%pn_opt => parse_node_get_next_ptr (pn_cmd)
  call cmd%compile_options (global)
  if (associated (pn_arg)) then
     n_lib = parse_node_get_n_sub (pn_arg)
  else
     n_lib = 0
  end if
  if (n_lib > 0) then
     allocate (cmd%libname (n_lib))
     pn_lib => parse_node_get_sub_ptr (pn_arg)
     do i = 1, n_lib
        cmd%libname(i) = parse_node_get_string (pn_lib)
        pn_lib => parse_node_get_next_ptr (pn_lib)
     end do
  end if
  if (associated (pn_exec_name)) then
     cmd%make_executable = .true.
     cmd%exec_name = parse_node_get_string (pn_exec_name)
  end if
end subroutine cmd_compile_compile
```

Command execution. Generate code, write driver, compile and link. Do this for all libraries in the list.

If no library names have been given and stored while compiling this command, we collect all libraries from the current stack and compile those.

We can compile static libraries (which actually just loads them). However, we can't incorporate in a generated executable.

⟨*Commands: cmd compile: TBP*⟩+≡

```
procedure :: execute => cmd_compile_execute
```

3018

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_execute (cmd, global)
      class(cmd_compile_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(string_t), dimension(:), allocatable :: libname, libname_static
      integer :: i
      if (allocated (cmd%libname)) then
         allocate (libname (size (cmd%libname)))
         libname = cmd%libname
      else
         call cmd%local%prclib_stack%get_names (libname)
      end if
      if (cmd%make_executable) then
         call get_prclib_static (libname_static)
         do i = 1, size (libname)
            if (any (libname_static == libname(i))) then
               call msg_fatal ("Compile: can't include static library '" &
                    // char (libname(i)) // "'")
            end if
         end do
         call compile_executable (cmd%exec_name, libname, cmd%local)
      else
         do i = 1, size (libname)
            call compile_library (libname(i), cmd%local)
         end do
      end if
    end subroutine cmd_compile_execute
```

This is the interface to the external procedure which returns the names of all static libraries which are part of the executable. (The default is none.) The routine must allocate the array.

⟨*Commands: public*⟩≡
```
    public :: get_prclib_static
```
⟨*Commands: interfaces*⟩+≡
```
    interface
       subroutine get_prclib_static (libname)
         import
         type(string_t), dimension(:), intent(inout), allocatable :: libname
       end subroutine get_prclib_static
    end interface
```

Execute a shell command

The argument is a string expression.
⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_exec_t
       private
       type(parse_node_t), pointer :: pn_command => null ()
     contains
     ⟨Commands: cmd exec: TBP⟩
    end type cmd_exec_t
```

Simply tell the status.

⟨Commands: cmd exec: TBP⟩≡
```
    procedure :: write => cmd_exec_write
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_exec_write (cmd, unit, indent)
      class(cmd_exec_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      if (associated (cmd%pn_command)) then
         write (u, "(1x,A)")  "exec: [command associated]"
      else
         write (u, "(1x,A)")  "exec: [undefined]"
      end if
    end subroutine cmd_exec_write
```

Compile the exec command.

⟨Commands: cmd exec: TBP⟩+≡
```
    procedure :: compile => cmd_exec_compile
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_exec_compile (cmd, global)
      class(cmd_exec_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_command
      pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
      pn_command => parse_node_get_sub_ptr (pn_arg)
      cmd%pn_command => pn_command
    end subroutine cmd_exec_compile
```

Execute the specified shell command.

⟨Commands: cmd exec: TBP⟩+≡
```
    procedure :: execute => cmd_exec_execute
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_exec_execute (cmd, global)
      class(cmd_exec_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(string_t) :: command
      logical :: is_known
      integer :: status
      command = eval_string (cmd%pn_command, global%var_list, is_known=is_known)
      if (is_known) then
         if (command /= "") then
            call os_system_call (command, status, verbose=.true.)
            if (status /= 0) then
               write (msg_buffer, "(A,I0)")  "Return code = ", status
               call msg_message ()
               call msg_error ("System command returned with nonzero status code")
            end if
         end if
      end if
```

```
      end subroutine cmd_exec_execute
```

Variable declaration

A variable can have various types. Hold the definition as an eval tree.

There are intrinsic variables, user variables, and model variables. The latter are further divided in independent variables and dependent variables.

Regarding model variables: When dealing with them, we always look at two variable lists in parallel. The global (or local) variable list contains the user-visible values. It includes variables that correspond to variables in the current model's list. These, in turn, are pointers to the model's parameter list, so the model is always in sync, internally. To keep the global variable list in sync with the model, the global variables carry the `is_copy` property and contain a separate pointer to the model variable. (The pointer is reassigned whenever the model changes.) Modifying the global variable changes two values simultaneously: the visible value and the model variable, via this extra pointer. After each modification, we update dependent parameters in the model variable list and re-synchronize the global variable list (again, using these pointers) with the model variable this. In the last step, modifications in the derived parameters become visible.

When we integrate a process, we capture the current variable list of the current model in a separate model instance, which is stored in the process object. Thus, the model parameters associated to this process at this time are preserved for the lifetime of the process object.

When we generate or rescan events, we can again capture a local model variable list in a model instance. This allows us to reweight event by event with different parameter sets simultaneously.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_var_t
      private
      type(string_t) :: name
      integer :: type = V_NONE
      type(parse_node_t), pointer :: pn_value => null ()
      logical :: is_intrinsic = .false.
      logical :: is_copy = .false.
    contains
    ⟨Commands: cmd var: TBP⟩
    end type cmd_var_t
```

Output. We know name, type, and properties, but not the value.

⟨*Commands: cmd var: TBP*⟩≡
```
    procedure :: write => cmd_var_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_var_write (cmd, unit, indent)
      class(cmd_var_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
```

```
      write (u, "(1x,A,A,A)", advance="no")  "var: ", char (cmd%name), " ("
      select case (cmd%type)
      case (V_NONE)
         write (u, "(A)", advance="no")  "[unknown]"
      case (V_LOG)
         write (u, "(A)", advance="no")  "logical"
      case (V_INT)
         write (u, "(A)", advance="no")  "int"
      case (V_REAL)
         write (u, "(A)", advance="no")  "real"
      case (V_CMPLX)
         write (u, "(A)", advance="no")  "complex"
      case (V_STR)
         write (u, "(A)", advance="no")  "string"
      case (V_PDG)
         write (u, "(A)", advance="no")  "alias"
      end select
      if (cmd%is_intrinsic) then
         write (u, "(A)", advance="no")  ", intrinsic"
      end if
      if (cmd%is_copy) then
         write (u, "(A)", advance="no")  ", copy"
      end if
      write (u, "(A)")  ")"
   end subroutine cmd_var_write
```

Compile the lhs and determine the variable name and type. Check whether this variable can be created or modified as requested, and append the value to the variable list, if appropriate. The value is initially undefined. The rhs is assigned to a pointer, to be compiled and evaluated when the command is executed.

⟨*Commands: cmd var: TBP*⟩+≡
```
      procedure :: compile => cmd_var_compile
```

⟨*Commands: procedures*⟩+≡
```
      subroutine cmd_var_compile (cmd, global)
        class(cmd_var_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(parse_node_t), pointer :: pn_var, pn_name
        type(parse_node_t), pointer :: pn_result, pn_proc
        type(string_t) :: var_name
        type(var_entry_t), pointer :: var_entry
        integer :: type
        logical :: new
        pn_result => null ()
        new = .false.
        select case (char (parse_node_get_rule_key (cmd%pn)))
        case ("cmd_log_decl");    type = V_LOG
           pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
           if (.not. associated (pn_var)) then   ! handle masked syntax error
              cmd%type = V_NONE; return
           end if
           pn_name => parse_node_get_sub_ptr (pn_var, 2)
           new = .true.
        case ("cmd_log");         type = V_LOG
```

```
      pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
case ("cmd_int");         type = V_INT
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
   new = .true.
case ("cmd_real");        type = V_REAL
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
   new = .true.
case ("cmd_complex");     type = V_CMPLX
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
   new = .true.
case ("cmd_num");         type = V_NONE
   pn_name => parse_node_get_sub_ptr (cmd%pn)
case ("cmd_string_decl"); type = V_STR
   pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
   if (.not. associated (pn_var)) then   ! handle masked syntax error
      cmd%type = V_NONE; return
   end if
   pn_name => parse_node_get_sub_ptr (pn_var, 2)
   new = .true.
case ("cmd_string");      type = V_STR
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
case ("cmd_alias");       type = V_PDG
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
   new = .true.
case ("cmd_result");      type = V_REAL
   pn_name => parse_node_get_sub_ptr (cmd%pn)
   pn_result => parse_node_get_sub_ptr (pn_name)
   pn_proc => parse_node_get_next_ptr (pn_result)
case default
   call parse_node_mismatch &
        ("logical|int|real|complex|?|$|alias|var_name", cmd%pn)  ! $
end select
if (.not. associated (pn_name)) then   ! handle masked syntax error
   cmd%type = V_NONE; return
end if
if (.not. associated (pn_result)) then
   var_name = parse_node_get_string (pn_name)
else
   var_name = parse_node_get_key (pn_result) &
        // "(" // parse_node_get_string (pn_proc) // ")"
end if
select case (type)
case (V_LOG);  var_name = "?" // var_name
case (V_STR);  var_name = "$" // var_name    ! $
end select
call var_list_check_user_var (global%var_list, var_name, type, new)
cmd%type = type
cmd%name = var_name
var_entry => var_list_get_var_ptr &
   (global%var_list, cmd%name, cmd%type, follow_link=.false.)
cmd%pn_value => parse_node_get_next_ptr (pn_name, 2)
if (.not. associated (cmd%pn_value))  cmd%type = V_NONE
if (associated (var_entry)) then
   cmd%is_intrinsic = var_entry_is_intrinsic (var_entry)
```

```
            cmd%is_copy = var_entry_is_copy (var_entry)
        else
            var_entry => var_list_get_var_ptr &
                (global%var_list, cmd%name, cmd%type, follow_link=.true.)
            if (associated (var_entry)) then
                cmd%is_intrinsic = var_entry_is_intrinsic (var_entry)
                if (var_entry_is_copy (var_entry)) then
                    cmd%is_copy = .true.
                    call var_list_init_copy (global%var_list, var_entry, user=.true.)
                end if
            end if
            if (.not. cmd%is_copy) then
                select case (cmd%type)
                case (V_LOG)
                    call var_list_append_log (global%var_list, cmd%name, &
                            intrinsic=cmd%is_intrinsic, user=.true.)
                case (V_INT)
                    call var_list_append_int (global%var_list, cmd%name, &
                            intrinsic=cmd%is_intrinsic, user=.true.)
                case (V_REAL)
                    call var_list_append_real (global%var_list, cmd%name, &
                            intrinsic=cmd%is_intrinsic, user=.true.)
                case (V_CMPLX)
                    call var_list_append_cmplx (global%var_list, cmd%name, &
                            intrinsic=cmd%is_intrinsic, user=.true.)
                case (V_PDG)
                    call var_list_append_pdg_array (global%var_list, cmd%name, &
                            intrinsic=cmd%is_intrinsic, user=.true.)
                case (V_STR)
                    call var_list_append_string (global%var_list, cmd%name, &
                            intrinsic=cmd%is_intrinsic, user=.true.)
                end select
            end if
        end if
    end subroutine cmd_var_compile
```

Execute. Evaluate the definition and assign the variable value. If the variable is a copy, the original is a model variable. The original is set automatically, and an update of the dependent parameters is in order.

⟨*Commands: cmd var: TBP*⟩+≡
```
    procedure :: execute => cmd_var_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_var_execute (cmd, global)
      class(cmd_var_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(string_t) :: model_name
      type(var_list_t), pointer :: model_vars
      if (associated (global%model)) then
         model_name = global%model%get_name ()
         model_vars => model_get_var_list_ptr (global%model)
         if (cmd%is_copy) then
            call var_list_set_original_pointer (global%var_list, cmd%name, &
                 model_vars)
```

```
            end if
            call cmd%set_value (global%var_list, &
                 verbose=.true., model_name=model_name)
            if (cmd%is_copy) then
               call var_list_restore (global%var_list)
               call model_parameters_update (global%model)
               call var_list_synchronize (global%var_list, model_vars)
            end if
         else
            call cmd%set_value (global%var_list, verbose=.true.)
         end if
      end subroutine cmd_var_execute
```

Copy the value to the variable list, where the variable should already exist.

⟨*Commands: cmd var: TBP*⟩+≡
```
      procedure :: set_value => cmd_var_set_value
```

⟨*Commands: procedures*⟩+≡
```
      subroutine cmd_var_set_value (var, var_list, verbose, model_name)
        class(cmd_var_t), intent(inout) :: var
        type(var_list_t), intent(inout), target :: var_list
        logical, intent(in), optional :: verbose
        type(string_t), intent(in), optional :: model_name
        logical :: lval, pacified
        integer :: ival
        real(default) :: rval
        complex(default) :: cval
        type(pdg_array_t) :: aval
        type(string_t) :: sval
        logical :: is_known
        pacified = var_list_get_lval (var_list, var_str ("?pacify"))
        select case (var%type)
        case (V_LOG)
           lval = eval_log (var%pn_value, var_list, is_known=is_known)
           call var_list_set_log (var_list, var%name, &
                lval, is_known, verbose=verbose, model_name=model_name)
        case (V_INT)
           ival = eval_int (var%pn_value, var_list, is_known=is_known)
           call var_list_set_int (var_list, var%name, &
                ival, is_known, verbose=verbose, model_name=model_name)
        case (V_REAL)
           rval = eval_real (var%pn_value, var_list, is_known=is_known)
           call var_list_set_real (var_list, var%name, &
                rval, is_known, verbose=verbose, &
                model_name=model_name, pacified = pacified)
        case (V_CMPLX)
           cval = eval_cmplx (var%pn_value, var_list, is_known=is_known)
           call var_list_set_cmplx (var_list, var%name, &
                cval, is_known, verbose=verbose, &
                model_name=model_name, pacified = pacified)
        case (V_PDG)
           aval = eval_pdg_array (var%pn_value, var_list, is_known=is_known)
           call var_list_set_pdg_array (var_list, var%name, &
                aval, is_known, verbose=verbose, model_name=model_name)
```

```
      case (V_STR)
         sval = eval_string (var%pn_value, var_list, is_known=is_known)
         call var_list_set_string (var_list, var%name, &
               sval, is_known, verbose=verbose, model_name=model_name)
      end select
   end subroutine cmd_var_set_value
```

SLHA

Read a SLHA (SUSY Les Houches Accord) file to fill the appropriate model parameters. We do not access the current variable record, but directly work on the appropriate SUSY model, which is loaded if necessary.

We may be in read or write mode. In the latter case, we may write just input parameters, or the complete spectrum, or the spectrum with all decays.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_slha_t
     private
     type(string_t) :: file
     logical :: write_mode = .false.
    contains
    ⟨Commands: cmd slha: TBP⟩
   end type cmd_slha_t
```

Output.

⟨*Commands: cmd slha: TBP*⟩≡
```
   procedure :: write => cmd_slha_write
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_slha_write (cmd, unit, indent)
     class(cmd_slha_t), intent(in) :: cmd
     integer, intent(in), optional :: unit, indent
     integer :: u
     u = output_unit (unit);  if (u < 0)  return
     call write_indent (u, indent)
     write (u, "(1x,A,A)")  "slha: file name  = ", char (cmd%file)
     write (u, "(1x,A,L1)") "slha: write mode = ", cmd%write_mode
   end subroutine cmd_slha_write
```

Compile. Read the filename and store it.

⟨*Commands: cmd slha: TBP*⟩+≡
```
   procedure :: compile => cmd_slha_compile
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_slha_compile (cmd, global)
     class(cmd_slha_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), pointer :: pn_key, pn_arg, pn_file
     pn_key => parse_node_get_sub_ptr (cmd%pn)
     pn_arg => parse_node_get_next_ptr (pn_key)
     pn_file => parse_node_get_sub_ptr (pn_arg)
     call cmd%compile_options (global)
     cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
```

```
      select case (char (parse_node_get_key (pn_key)))
      case ("read_slha")
         cmd%write_mode = .false.
      case ("write_slha")
         cmd%write_mode = .true.
      case default
         call parse_node_mismatch ("read_slha|write_slha",  cmd%pn)
      end select
      cmd%file = parse_node_get_string (pn_file)
   end subroutine cmd_slha_compile
```

Execute. Read or write the specified SLHA file. Behind the scenes, this will first read the WHIZARD model file, then read the SLHA file and assign the SLHA parameters as far as determined by `dispatch_slha`. Finally, the global variables are synchronized with the model. This is similar to executing `cmd_model`.

⟨*Commands: cmd slha: TBP*⟩+≡
```
   procedure :: execute => cmd_slha_execute
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_slha_execute (cmd, global)
     class(cmd_slha_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     logical :: input, spectrum, decays
     type(var_list_t), pointer :: model_vars
     if (cmd%write_mode) then
        input = .true.
        spectrum = .false.
        decays = .false.
        call slha_write_file &
             (cmd%file, cmd%local%model, &
              input = input, spectrum = spectrum, decays = decays)
     else
        call dispatch_slha (cmd%local, &
             input = input, spectrum = spectrum, decays = decays)
        call slha_read_file &
             (cmd%file, cmd%local%os_data, global%model_list, global%model, &
              input = input, spectrum = spectrum, decays = decays)
        model_vars => model_get_var_list_ptr (global%model)
        call var_list_synchronize (global%var_list, model_vars, &
             reset_pointers = .true.)
        call global%particle_stack%reset (global%model)
     end if
   end subroutine cmd_slha_execute
```

Show values

This command shows the current values of variables or other objects, in a suitably condensed form.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_show_t
     private
     type(string_t), dimension(:), allocatable :: name
```

```
  contains
⟨Commands: cmd show: TBP⟩
end type cmd_show_t
```

Output: list the object names, not values.

⟨*Commands: cmd show: TBP*⟩≡
```
    procedure :: write => cmd_show_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_show_write (cmd, unit, indent)
      class(cmd_show_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "show: "
      if (allocated (cmd%name)) then
         do i = 1, size (cmd%name)
            write (u, "(1x,A)", advance="no")  char (cmd%name(i))
         end do
         write (u, *)
      else
         write (u, "(5x,A)")  "[undefined]"
      end if
    end subroutine cmd_show_write
```

Compile. Allocate an array which is filled with the names of the variables to show.

⟨*Commands: cmd show: TBP*⟩+≡
```
    procedure :: compile => cmd_show_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_show_compile (cmd, global)
      class(cmd_show_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_var, pn_prefix, pn_name
      type(string_t) :: key
      integer :: i, n_args
      pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
      if (associated (pn_arg)) then
         select case (char (parse_node_get_rule_key (pn_arg)))
         case ("show_arg")
            cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
         case default
            cmd%pn_opt => pn_arg
            pn_arg => null ()
         end select
      end if
      call cmd%compile_options (global)
      if (associated (pn_arg)) then
         n_args = parse_node_get_n_sub (pn_arg)
         allocate (cmd%name (n_args))
         pn_var => parse_node_get_sub_ptr (pn_arg)
         i = 0
```

```
                do while (associated (pn_var))
                   i = i + 1
                   select case (char (parse_node_get_rule_key (pn_var)))
                   case ("model", "library", "beams", "iterations", &
                         "cuts", "weight", "int", "real", "complex", &
                         "scale", "factorization_scale", "renormalization_scale", &
                         "selection", "reweight", "analysis", "pdg", &
                         "stable", "unstable", "polarized", "unpolarized", &
                         "results", "expect", "intrinsic", "string", "logical")
                      cmd%name(i) = parse_node_get_key (pn_var)
                   case ("result_var")
                      pn_prefix => parse_node_get_sub_ptr (pn_var)
                      pn_name => parse_node_get_next_ptr (pn_prefix)
                      if (associated (pn_name)) then
                         cmd%name(i) = parse_node_get_key (pn_prefix) &
                               // "(" // parse_node_get_string (pn_name) // ")"
                      else
                         cmd%name(i) = parse_node_get_key (pn_prefix)
                      end if
                   case ("log_var", "string_var", "alias_var")
                      pn_prefix => parse_node_get_sub_ptr (pn_var)
                      pn_name => parse_node_get_next_ptr (pn_prefix)
                      key = parse_node_get_key (pn_prefix)
                      if (associated (pn_name)) then
                         select case (char (parse_node_get_rule_key (pn_name)))
                         case ("var_name")
                            select case (char (key))
                            case ("?", "$")  ! $ sign
                               cmd%name(i) = key // parse_node_get_string (pn_name)
                            case ("alias")
                               cmd%name(i) = parse_node_get_string (pn_name)
                            end select
                         case default
                            call parse_node_mismatch &
                                  ("var_name",  pn_name)
                         end select
                      else
                         cmd%name(i) = key
                      end if
                   case default
                      cmd%name(i) = parse_node_get_string (pn_var)
                   end select
                   pn_var => parse_node_get_next_ptr (pn_var)
                end do
           else
              allocate (cmd%name (0))
           end if
    end subroutine cmd_show_compile
```

Execute. Scan the list of objects to show.

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: SHOW_BUFFER_SIZE = 4096
```

⟨*Commands: cmd show: TBP*⟩+≡

```
              procedure :: execute => cmd_show_execute
⟨Commands: procedures⟩+≡
      subroutine cmd_show_execute (cmd, global)
        class(cmd_show_t), intent(inout) :: cmd
        type(rt_data_t), intent(inout), target :: global
        type(model_t), pointer :: model
        type(string_t) :: name
        integer :: n, pdg
        type(flavor_t) :: flv
        type(process_library_t), pointer :: prc_lib
        type(process_t), pointer :: process
        logical :: pacified
        character(SHOW_BUFFER_SIZE) :: buffer
        integer :: i, j, u, u_log, u_out
        u = free_unit ()
        pacified = var_list_get_lval (cmd%local%var_list, &
             var_str ("?pacify"))
        open (u, status = "scratch", action = "readwrite")
        if (associated (cmd%local%model)) then
           name = cmd%local%model%get_name ()
        end if
        if (size (cmd%name) == 0) then
           if (associated (cmd%local%model)) then
              call var_list_write (cmd%local%var_list, model_name = name, &
                   unit = u, pacified = pacified)
           else
              call var_list_write (cmd%local%var_list, unit = u, pacified = pacified)
           end if
        else
           do i = 1, size (cmd%name)
              select case (char (cmd%name(i)))
              case ("model")
                 if (associated (cmd%local%model)) then
                    call model_show (cmd%local%model, u)
                 else
                    write (u, "(A)")  "Model: [undefined]"
                 end if
              case ("library")
                 if (associated (cmd%local%prclib)) then
                    call cmd%local%prclib%show (u)
                 else
                    write (u, "(A)")  "Process library: [undefined]"
                 end if
              case ("beams")
                 call cmd%local%show_beams (u)
              case ("iterations")
                 call cmd%local%it_list%write (u)
              case ("results")
                 call cmd%local%process_stack%show (u)
              case ("stable")
                 call model_show_stable (cmd%local%model, u)
              case ("polarized")
                 call model_show_polarized (cmd%local%model, u)
              case ("unpolarized")
```

3030

```
         call model_show_unpolarized (cmd%local%model, u)
case ("unstable")
   model => cmd%local%model
   call model_show_unstable (model, u)
   n = model_get_n_particles (model)
   do j = 1, n
      pdg = model_get_particle_pdg (model, j)
      call flavor_init (flv, pdg, model)
      if (.not. flavor_is_stable (flv)) &
           call show_unstable (cmd%local, pdg, u)
      if (flavor_has_antiparticle (flv)) then
         if (.not. flavor_is_stable (flavor_anti (flv))) &
              call show_unstable (cmd%local, -pdg, u)
      end if
   end do
case ("cuts", "weight", "scale", &
      "factorization_scale", "renormalization_scale", &
      "selection", "reweight", "analysis")
   call cmd%local%pn%show (cmd%name(i), u)
case ("expect")
   call expect_summary (force = .true.)
case ("logical")
   if (associated (cmd%local%model)) then
      call var_list_write (cmd%local%var_list, only_type=V_LOG, &
           model_name = name, unit=u, pacified = pacified)
   else
      call var_list_write (cmd%local%var_list, &
           only_type=V_LOG, unit=u, pacified = pacified)
   end if
case ("intrinsic")
   if (associated (cmd%local%model)) then
      call var_list_write (cmd%local%var_list, intrinsic=.true., &
           model_name = name, unit=u, pacified = pacified)
   else
      call var_list_write (cmd%local%var_list, &
           intrinsic=.true., unit=u, pacified = pacified)
   end if
case ("int")
   if (associated (cmd%local%model)) then
      call var_list_write (cmd%local%var_list, only_type=V_INT, &
           model_name = name, unit=u, pacified = pacified)
   else
      call var_list_write (cmd%local%var_list, only_type=V_INT, &
           unit=u, pacified = pacified)
   end if
case ("real")
   if (associated (cmd%local%model)) then
      call var_list_write (cmd%local%var_list, only_type=V_REAL, &
           model_name = name, unit=u, pacified = pacified)
   else
      call var_list_write (cmd%local%var_list, only_type=V_REAL, &
           unit=u, pacified = pacified)
   end if
case ("complex")
```

```fortran
            if (associated (cmd%local%model)) then
               call var_list_write (cmd%local%var_list, only_type=V_CMPLX, &
                     model_name = name, unit=u, pacified = pacified)
            else
               call var_list_write (cmd%local%var_list, only_type=V_CMPLX, &
                     unit=u, pacified = pacified)
            end if
         case ("pdg")
            if (associated (cmd%local%model)) then
               call var_list_write (cmd%local%var_list, only_type=V_PDG, &
                     model_name = name, unit=u, pacified = pacified)
            else
               call var_list_write (cmd%local%var_list, only_type=V_PDG, &
                     unit=u, pacified = pacified)
            end if
         case ("string")
            if (associated (global%model)) then
               call var_list_write (cmd%local%var_list, only_type=V_STR, &
                     model_name = name, unit=u, pacified = pacified)
            else
               call var_list_write (cmd%local%var_list, only_type=V_STR, &
                     unit=u, pacified = pacified)
            end if
         case default
            if (var_list_exists (cmd%local%var_list, cmd%name(i))) then
               call var_list_write_var (cmd%local%var_list, cmd%name(i), &
                     unit = u, pacified = pacified)
            else if (analysis_exists (cmd%name(i))) then
               call analysis_write (cmd%name(i), u)
            else if (cmd%local%process_stack%exists (cmd%name(i))) then
               process => cmd%local%process_stack%get_process_ptr (cmd%name(i))
               call process%show (u)
            else if (associated (cmd%local%prclib_stack%get_library_ptr &
               (cmd%name(i)))) then
               prc_lib => cmd%local%prclib_stack%get_library_ptr (cmd%name(i))
               call prc_lib%show (u)
            else
               call msg_error ("show: object '" // char (cmd%name(i)) &
                     // "' not found")
            end if
         end select
      end do
   end if
   rewind (u)
   u_log = logfile_unit ()
   u_out = output_unit ()
   do
      read (u, "(A)", end = 1)  buffer
      if (u_log > 0)  write (u_log, "(A)")  trim (buffer)
      if (u_out > 0)  write (u_out, "(A)")  trim (buffer)
   end do
1  close (u)
   if (u_log > 0)  flush (u_log)
   if (u_out > 0)  flush (u_out)
```

```
      end subroutine cmd_show_execute
```

Clear values

This command clears the current values of variables or other objects, where this
makes sense. It parallels the show command. The objects are cleared, but not
deleted.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_clear_t
      private
      type(string_t), dimension(:), allocatable :: name
     contains
     ⟨Commands: cmd clear: TBP⟩
    end type cmd_clear_t
```

Output: list the names of the objects to be cleared.

⟨*Commands: cmd clear: TBP*⟩≡
```
    procedure :: write => cmd_clear_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_clear_write (cmd, unit, indent)
      class(cmd_clear_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "clear: "
      if (allocated (cmd%name)) then
         do i = 1, size (cmd%name)
            write (u, "(1x,A)", advance="no")  char (cmd%name(i))
         end do
         write (u, *)
      else
         write (u, "(5x,A)")  "[undefined]"
      end if
    end subroutine cmd_clear_write
```

Compile. Allocate an array which is filled with the names of the objects to be
cleared.

    Note: there is currently no need to account for options, but we prepare for
that possibility.

⟨*Commands: cmd clear: TBP*⟩+≡
```
    procedure :: compile => cmd_clear_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_clear_compile (cmd, global)
      class(cmd_clear_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_var, pn_prefix, pn_name
      type(string_t) :: key
      integer :: i, n_args
      pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
```

```
   if (associated (pn_arg)) then
      select case (char (parse_node_get_rule_key (pn_arg)))
      case ("clear_arg")
         cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
      case default
         cmd%pn_opt => pn_arg
         pn_arg => null ()
      end select
   end if
   call cmd%compile_options (global)
   if (associated (pn_arg)) then
      n_args = parse_node_get_n_sub (pn_arg)
      allocate (cmd%name (n_args))
      pn_var => parse_node_get_sub_ptr (pn_arg)
      i = 0
      do while (associated (pn_var))
         i = i + 1
         select case (char (parse_node_get_rule_key (pn_var)))
         case ("beams", "iterations", &
               "cuts", "weight", &
               "scale", "factorization_scale", "renormalization_scale", &
               "selection", "reweight", "analysis", &
               "unstable", "polarized", &
               "expect")
            cmd%name(i) = parse_node_get_key (pn_var)
         case ("log_var", "string_var")
            pn_prefix => parse_node_get_sub_ptr (pn_var)
            pn_name => parse_node_get_next_ptr (pn_prefix)
            key = parse_node_get_key (pn_prefix)
            if (associated (pn_name)) then
               select case (char (parse_node_get_rule_key (pn_name)))
               case ("var_name")
                  select case (char (key))
                  case ("?", "$")  ! $ sign
                     cmd%name(i) = key // parse_node_get_string (pn_name)
                  end select
               case default
                  call parse_node_mismatch &
                       ("var_name",  pn_name)
               end select
            else
               cmd%name(i) = key
            end if
         case default
            cmd%name(i) = parse_node_get_string (pn_var)
         end select
         pn_var => parse_node_get_next_ptr (pn_var)
      end do
   else
      allocate (cmd%name (0))
   end if
 end subroutine cmd_clear_compile
```

Execute. Scan the list of objects to clear.

Objects that can be shown but not cleared: model, library, results

⟨*Commands: cmd clear: TBP*⟩+≡

```
    procedure :: execute => cmd_clear_execute
```

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_clear_execute (cmd, global)
      class(cmd_clear_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(var_entry_t), pointer :: var
      integer :: i
      logical :: success
      if (size (cmd%name) == 0) then
         call msg_warning ("clear: no object specified")
      else
         do i = 1, size (cmd%name)
            success = .true.
            select case (char (cmd%name(i)))
            case ("beams")
               call cmd%local%clear_beams ()
            case ("iterations")
               call cmd%local%it_list%clear ()
            case ("polarized")
               call model_clear_polarized (cmd%local%model)
            case ("unstable")
               call model_clear_unstable (cmd%local%model)
            case ("cuts", "weight", "scale", &
                 "factorization_scale", "renormalization_scale", &
                 "selection", "reweight", "analysis")
               call cmd%local%pn%clear (cmd%name(i))
            case ("expect")
               call expect_clear ()
            case default
               if (var_list_exists (cmd%local%var_list, cmd%name(i))) then
                  var => var_list_get_var_ptr &
                       (cmd%local%var_list, cmd%name(i), follow_link=.true.)
                  if (.not. var_entry_is_locked (var)) then
                     call var_entry_clear (var)
                  else
                     call msg_error ("clear: variable '" // char (cmd%name(i)) &
                          // "' is locked and can't be cleared")
                     success = .false.
                  end if
               else if (analysis_exists (cmd%name(i))) then
                  call analysis_clear (cmd%name(i))
               else
                  call msg_error ("clear: object '" // char (cmd%name(i)) &
                       // "' not found")
                  success = .false.
               end if
            end select
            if (success)  call msg_message ("cleared: " // char (cmd%name(i)))
         end do
      end if
    end subroutine cmd_clear_execute
```

Compare values of variables to expectation

The implementation is similar to the `show` command. There are just two arguments: two values that should be compared. For providing local values for the numerical tolerance, the command has a local argument list.

    If the expectation fails, an error condition is recorded.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_expect_t
   private
   type(parse_node_t), pointer :: pn_lexpr => null ()
 contains
 ⟨Commands: cmd expect: TBP⟩
 end type cmd_expect_t
```

Simply tell the status.

⟨*Commands: cmd expect: TBP*⟩≡
```
procedure :: write => cmd_expect_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_expect_write (cmd, unit, indent)
  class(cmd_expect_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  if (associated (cmd%pn_lexpr)) then
     write (u, "(1x,A)")  "expect: [expression associated]"
  else
     write (u, "(1x,A)")  "expect: [undefined]"
  end if
end subroutine cmd_expect_write
```

Compile. This merely assigns the parse node, the actual compilation is done at execution. This is necessary because the origin of variables (local/global) may change during execution.

⟨*Commands: cmd expect: TBP*⟩+≡
```
procedure :: compile => cmd_expect_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_expect_compile (cmd, global)
  class(cmd_expect_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg
  pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
  cmd%pn_lexpr => parse_node_get_sub_ptr (pn_arg)
  call cmd%compile_options (global)
end subroutine cmd_expect_compile
```

Execute. Evaluate both arguments, print them and their difference (if numerical), and whether they agree. Record the result.

⟨*Commands: cmd expect: TBP*⟩+≡
```
procedure :: execute => cmd_expect_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_expect_execute (cmd, global)
      class(cmd_expect_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      logical :: success, is_known
      success = &
           eval_log (cmd%pn_lexpr, cmd%local%var_list, is_known=is_known)
      if (is_known) then
         if (success) then
            call msg_message ("expect: success")
         else
            call msg_error ("expect: failure")
         end if
      else
         call msg_error ("expect: undefined result")
         success = .false.
      end if
      call expect_record (success)
    end subroutine cmd_expect_execute
```

Beams

The beam command includes both beam and structure-function definition.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_beams_t
      private
      integer :: n_in = 0
      type(parse_node_p), dimension(:), allocatable :: pn_pdg
      integer :: n_sf_record = 0
      integer, dimension(:), allocatable :: n_entry
      type(parse_node_p), dimension(:,:), allocatable :: pn_sf_entry
     contains
     ⟨Commands: cmd beams: TBP⟩
    end type cmd_beams_t
```

Output. The particle expressions are not resolved.

⟨*Commands: cmd beams: TBP*⟩≡
```
    procedure :: write => cmd_beams_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_write (cmd, unit, indent)
      class(cmd_beams_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      select case (cmd%n_in)
      case (1)
         write (u, "(1x,A)")  "beams: 1 [decay]"
      case (2)
         write (u, "(1x,A)")  "beams: 2 [scattering]"
      case default
```

```
           write (u, "(1x,A)")  "beams: [undefined]"
        end select
        if (allocated (cmd%n_entry)) then
           if (cmd%n_sf_record > 0) then
              write (u, "(1x,A,99(1x,I0))")  "structure function entries:", &
                    cmd%n_entry
           end if
        end if
     end subroutine cmd_beams_write
```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

⟨*Commands: cmd beams: TBP*⟩+≡
```
     procedure :: compile => cmd_beams_compile
```

⟨*Commands: procedures*⟩+≡
```
     subroutine cmd_beams_compile (cmd, global)
       class(cmd_beams_t), intent(inout) :: cmd
       type(rt_data_t), intent(inout), target :: global
       type(parse_node_t), pointer :: pn_beam_def, pn_beam_spec
       type(parse_node_t), pointer :: pn_beam_list
       type(parse_node_t), pointer :: pn_codes
       type(parse_node_t), pointer :: pn_strfun_seq, pn_strfun_pair
       type(parse_node_t), pointer :: pn_strfun_def
       integer :: i
       pn_beam_def => parse_node_get_sub_ptr (cmd%pn, 3)
       pn_beam_spec => parse_node_get_sub_ptr (pn_beam_def)
       pn_strfun_seq => parse_node_get_next_ptr (pn_beam_spec)
       pn_beam_list => parse_node_get_sub_ptr (pn_beam_spec)
       call cmd%compile_options (global)
       cmd%n_in = parse_node_get_n_sub (pn_beam_list)
       allocate (cmd%pn_pdg (cmd%n_in))
       pn_codes => parse_node_get_sub_ptr (pn_beam_list)
       do i = 1, cmd%n_in
          cmd%pn_pdg(i)%ptr => pn_codes
          pn_codes => parse_node_get_next_ptr (pn_codes)
       end do
       if (associated (pn_strfun_seq)) then
          cmd%n_sf_record = parse_node_get_n_sub (pn_beam_def) - 1
          allocate (cmd%n_entry (cmd%n_sf_record), source = 1)
          allocate (cmd%pn_sf_entry (2, cmd%n_sf_record))
          do i = 1, cmd%n_sf_record
             pn_strfun_pair => parse_node_get_sub_ptr (pn_strfun_seq, 2)
             pn_strfun_def => parse_node_get_sub_ptr (pn_strfun_pair)
             cmd%pn_sf_entry(1,i)%ptr => pn_strfun_def
             pn_strfun_def => parse_node_get_next_ptr (pn_strfun_def)
             cmd%pn_sf_entry(2,i)%ptr => pn_strfun_def
             if (associated (pn_strfun_def))  cmd%n_entry(i) = 2
             pn_strfun_seq => parse_node_get_next_ptr (pn_strfun_seq)
          end do
       else
          allocate (cmd%n_entry (0))
          allocate (cmd%pn_sf_entry (0, 0))
       end if
```

```
      end subroutine cmd_beams_compile
```

Command execution: Determine beam particles and structure-function names,
if any. The results are stored in the `beam_structure` component of the `global`
data block.

⟨*Commands: cmd beams: TBP*⟩+≡
```
    procedure :: execute => cmd_beams_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_execute (cmd, global)
      class(cmd_beams_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(pdg_array_t) :: pdg_array
      integer, dimension(:), allocatable :: pdg
      type(flavor_t), dimension(:), allocatable :: flv
      type(parse_node_t), pointer :: pn_key
      type(string_t) :: sf_name
      integer :: i, j
      call lhapdf_status_reset (global%lhapdf_status)
      call pdf_builtin_status_reset (global%pdf_builtin_status)
      allocate (flv (cmd%n_in))
      do i = 1, cmd%n_in
         pdg_array = eval_pdg_array (cmd%pn_pdg(i)%ptr, cmd%local%var_list)
         pdg = pdg_array
         select case (size (pdg))
         case (1)
            call flavor_init (flv(i), pdg(1), cmd%local%model)
         case default
            call msg_fatal ("Beams: beam particles must be unique")
         end select
      end do
      select case (cmd%n_in)
      case (1)
         if (cmd%n_sf_record > 0) then
            call msg_fatal ("Beam setup: no structure functions allowed &
                 &for decay")
         end if
         call global%beam_structure%init_sf (flavor_get_name (flv))
      case (2)
         call global%beam_structure%init_sf (flavor_get_name (flv), cmd%n_entry)
         do i = 1, cmd%n_sf_record
            do j = 1, cmd%n_entry(i)
               pn_key => parse_node_get_sub_ptr (cmd%pn_sf_entry(j,i)%ptr)
               sf_name = parse_node_get_key (pn_key)
               call global%beam_structure%set_sf (i, j, sf_name)
            end do
         end do
      end select
    end subroutine cmd_beams_execute
```

Density matrices for beam polarization

For holding beam polarization, we define a notation and a data structure for sparse matrices. The entries (and the index expressions) are numerical expressions, so we use evaluation trees.

Each entry in the sparse matrix is an n-tuple of expressions. The first tuple elements represent index values, the last one is an arbitrary (complex) number. Absent expressions are replaced by default-value rules.

Note: Here, and in some other commands, we would like to store an evaluation tree, not just a parse node pointer. However, the current expression handler wants all variables defined, so the evaluation tree can only be built by `evaluate`, i.e., compiled just-in-time and evaluated immediately.

⟨*Commands: types*⟩+≡
```
type :: sentry_expr_t
   type(parse_node_p), dimension(:), allocatable :: expr
 contains
 ⟨Commands: sentry expr: TBP⟩
 end type sentry_expr_t
```

Compile parse nodes into evaluation trees.

⟨*Commands: sentry expr: TBP*⟩≡
```
procedure :: compile => sentry_expr_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine sentry_expr_compile (sentry, pn)
  class(sentry_expr_t), intent(out) :: sentry
  type(parse_node_t), intent(in), target :: pn
  type(parse_node_t), pointer :: pn_expr, pn_extra
  integer :: n_expr, i
  n_expr = parse_node_get_n_sub (pn)
  allocate (sentry%expr (n_expr))
  if (n_expr > 0) then
     i = 0
     pn_expr => parse_node_get_sub_ptr (pn)
     pn_extra => parse_node_get_next_ptr (pn_expr)
     do i = 1, n_expr
        sentry%expr(i)%ptr => pn_expr
        if (associated (pn_extra)) then
           pn_expr => parse_node_get_sub_ptr (pn_extra, 2)
           pn_extra => parse_node_get_next_ptr (pn_extra)
        end if
     end do
  end if
end subroutine sentry_expr_compile
```

Evaluate the expressions and return an index array of predefined length together with a complex value. If the value (as the last expression) is undefined, set it to unity. If index values are undefined, repeat the previous index value.

⟨*Commands: sentry expr: TBP*⟩+≡
```
procedure :: evaluate => sentry_expr_evaluate
```

⟨*Commands: procedures*⟩+≡
```
subroutine sentry_expr_evaluate (sentry, index, value, global)
```

```
      class(sentry_expr_t), intent(inout) :: sentry
      integer, dimension(:), intent(out) :: index
      complex(default), intent(out) :: value
      type(rt_data_t), intent(in), target :: global
      integer :: i, n_expr, n_index
      type(eval_tree_t) :: eval_tree
      n_expr = size (sentry%expr)
      n_index = size (index)
      if (n_expr <= n_index + 1) then
         do i = 1, min (n_expr, n_index)
            associate (expr => sentry%expr(i))
              call eval_tree_init_expr (eval_tree, expr%ptr, global%var_list)
              call eval_tree_evaluate (eval_tree)
              if (eval_tree_result_is_known (eval_tree)) then
                 index(i) = eval_tree_get_int (eval_tree)
              else
                 call msg_fatal ("Evaluating density matrix: undefined index")
              end if
            end associate
         end do
         do i = n_expr + 1, n_index
            index(i) = index(n_expr)
         end do
         if (n_expr == n_index + 1) then
            associate (expr => sentry%expr(n_expr))
              call eval_tree_init_expr (eval_tree, expr%ptr, global%var_list)
              call eval_tree_evaluate (eval_tree)
              if (eval_tree_result_is_known (eval_tree)) then
                 value = eval_tree_get_cmplx (eval_tree)
              else
                 call msg_fatal ("Evaluating density matrix: undefined index")
              end if
              call eval_tree_final (eval_tree)
            end associate
         else
              value = 1
         end if
      else
         call msg_fatal ("Evaluating density matrix: index expression too long")
      end if
    end subroutine sentry_expr_evaluate
```

The sparse matrix itself consists of an arbitrary number of entries.
⟨*Commands: types*⟩+≡
```
    type :: smatrix_expr_t
      type(sentry_expr_t), dimension(:), allocatable :: entry
     contains
     ⟨Commands: smatrix expr: TBP⟩
    end type smatrix_expr_t
```

Compile: assign sub-nodes to sentry-expressions and compile those.
⟨*Commands: smatrix expr: TBP*⟩≡
```
    procedure :: compile => smatrix_expr_compile
```

⟨*Commands: procedures*⟩+≡
```
  subroutine smatrix_expr_compile (smatrix_expr, pn)
    class(smatrix_expr_t), intent(out) :: smatrix_expr
    type(parse_node_t), intent(in), target :: pn
    type(parse_node_t), pointer :: pn_arg, pn_entry
    integer :: n_entry, i
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    if (associated (pn_arg)) then
       n_entry = parse_node_get_n_sub (pn_arg)
       allocate (smatrix_expr%entry (n_entry))
       pn_entry => parse_node_get_sub_ptr (pn_arg)
       do i = 1, n_entry
          call smatrix_expr%entry(i)%compile (pn_entry)
          pn_entry => parse_node_get_next_ptr (pn_entry)
       end do
    else
       allocate (smatrix_expr%entry (0))
    end if
  end subroutine smatrix_expr_compile
```

Evaluate the entries and build a new `smatrix` object, which contains just the numerical results.

⟨*Commands: smatrix expr: TBP*⟩+≡
```
  procedure :: evaluate => smatrix_expr_evaluate
```

⟨*Commands: procedures*⟩+≡
```
  subroutine smatrix_expr_evaluate (smatrix_expr, smatrix, global)
    class(smatrix_expr_t), intent(inout) :: smatrix_expr
    type(smatrix_t), intent(out) :: smatrix
    type(rt_data_t), intent(in), target :: global
    integer, dimension(2) :: idx
    complex(default) :: value
    integer :: i, n_entry
    n_entry = size (smatrix_expr%entry)
    call smatrix%init (2, n_entry)
    do i = 1, n_entry
       call smatrix_expr%entry(i)%evaluate (idx, value, global)
       call smatrix%set_entry (i, idx, value)
    end do
  end subroutine smatrix_expr_evaluate
```

Beam polarization density

The beam polarization command defines spin density matrix for one or two beams (scattering or decay).

⟨*Commands: types*⟩+≡
```
  type, extends (command_t) :: cmd_beams_pol_density_t
    private
    integer :: n_in = 0
    type(smatrix_expr_t), dimension(:), allocatable :: smatrix
   contains
    ⟨Commands: cmd beams pol density: TBP⟩
```

```
      end type cmd_beams_pol_density_t
```

Output.

⟨*Commands: cmd beams pol density: TBP*⟩≡
```
    procedure :: write => cmd_beams_pol_density_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_pol_density_write (cmd, unit, indent)
      class(cmd_beams_pol_density_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      select case (cmd%n_in)
      case (1)
         write (u, "(1x,A)")  "beams polarization setup: 1 [decay]"
      case (2)
         write (u, "(1x,A)")  "beams polarization setup: 2 [scattering]"
      case default
         write (u, "(1x,A)")  "beams polarization setup: [undefined]"
      end select
    end subroutine cmd_beams_pol_density_write
```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

⟨*Commands: cmd beams pol density: TBP*⟩+≡
```
    procedure :: compile => cmd_beams_pol_density_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_pol_density_compile (cmd, global)
      class(cmd_beams_pol_density_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_pol_spec, pn_smatrix
      integer :: i
      pn_pol_spec => parse_node_get_sub_ptr (cmd%pn, 3)
      call cmd%compile_options (global)
      cmd%n_in = parse_node_get_n_sub (pn_pol_spec)
      allocate (cmd%smatrix (cmd%n_in))
      pn_smatrix => parse_node_get_sub_ptr (pn_pol_spec)
      do i = 1, cmd%n_in
         call cmd%smatrix(i)%compile (pn_smatrix)
         pn_smatrix => parse_node_get_next_ptr (pn_smatrix)
      end do
    end subroutine cmd_beams_pol_density_compile
```

Command execution: Fill polarization density matrices. No check yet, the matrices are checked and normalized when the actual beam object is created, just before integration. For intermediate storage, we use the `beam_structure` object in the `global` data set.

⟨*Commands: cmd beams pol density: TBP*⟩+≡
```
    procedure :: execute => cmd_beams_pol_density_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_pol_density_execute (cmd, global)
      class(cmd_beams_pol_density_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(smatrix_t) :: smatrix
      integer :: i
      call global%beam_structure%init_pol (cmd%n_in)
      do i = 1, cmd%n_in
         call cmd%smatrix(i)%evaluate (smatrix, global)
         call global%beam_structure%set_smatrix (i, smatrix)
      end do
    end subroutine cmd_beams_pol_density_execute
```

Beam polarization fraction

In addition to the polarization density matrix, we can independently specify the polarization fraction for one or both beams.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_beams_pol_fraction_t
      private
      integer :: n_in = 0
      type(parse_node_p), dimension(:), allocatable :: expr
    contains
    ⟨Commands: cmd beams pol fraction: TBP⟩
    end type cmd_beams_pol_fraction_t
```

Output.

⟨*Commands: cmd beams pol fraction: TBP*⟩≡
```
    procedure :: write => cmd_beams_pol_fraction_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_pol_fraction_write (cmd, unit, indent)
      class(cmd_beams_pol_fraction_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      select case (cmd%n_in)
      case (1)
         write (u, "(1x,A)")  "beams polarization fraction: 1 [decay]"
      case (2)
         write (u, "(1x,A)")  "beams polarization fraction: 2 [scattering]"
      case default
         write (u, "(1x,A)")  "beams polarization fraction: [undefined]"
      end select
    end subroutine cmd_beams_pol_fraction_write
```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

⟨*Commands: cmd beams pol fraction: TBP*⟩+≡
```
    procedure :: compile => cmd_beams_pol_fraction_compile
```

3044

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_pol_fraction_compile (cmd, global)
      class(cmd_beams_pol_fraction_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_frac_spec, pn_expr
      integer :: i
      pn_frac_spec => parse_node_get_sub_ptr (cmd%pn, 3)
      call cmd%compile_options (global)
      cmd%n_in = parse_node_get_n_sub (pn_frac_spec)
      allocate (cmd%expr (cmd%n_in))
      pn_expr => parse_node_get_sub_ptr (pn_frac_spec)
      do i = 1, cmd%n_in
         cmd%expr(i)%ptr => pn_expr
         pn_expr => parse_node_get_next_ptr (pn_expr)
      end do
    end subroutine cmd_beams_pol_fraction_compile
```

Command execution: Retrieve the numerical values of the beam polarization fractions. The results are stored in the `beam_structure` component of the `global` data block.

⟨*Commands: cmd beams pol fraction: TBP*⟩+≡
```
    procedure :: execute => cmd_beams_pol_fraction_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_pol_fraction_execute (cmd, global)
      class(cmd_beams_pol_fraction_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      real(default), dimension(:), allocatable :: pol_f
      type(eval_tree_t) :: expr
      integer :: i
      allocate (pol_f (cmd%n_in))
      do i = 1, cmd%n_in
         call eval_tree_init_expr (expr, cmd%expr(i)%ptr, global%var_list)
         call eval_tree_evaluate (expr)
         if (eval_tree_result_is_known (expr)) then
            pol_f(i) = eval_tree_get_real (expr)
         else
            call msg_fatal ("beams polarization fraction: undefined value")
         end if
         call eval_tree_final (expr)
      end do
      call global%beam_structure%set_pol_f (pol_f)
    end subroutine cmd_beams_pol_fraction_execute
```

Beam momentum

This is completely analogous to the previous command, hence we can use inheritance.

⟨*Commands: types*⟩+≡
```
    type, extends (cmd_beams_pol_fraction_t) :: cmd_beams_momentum_t
     contains
```
⟨*Commands: cmd beams momentum: TBP*⟩

3045

```
    end type cmd_beams_momentum_t
```

Output.

⟨*Commands: cmd beams momentum: TBP*⟩≡
```
    procedure :: write => cmd_beams_momentum_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_momentum_write (cmd, unit, indent)
      class(cmd_beams_momentum_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      select case (cmd%n_in)
      case (1)
         write (u, "(1x,A)")  "beams momentum: 1 [decay]"
      case (2)
         write (u, "(1x,A)")  "beams momentum: 2 [scattering]"
      case default
         write (u, "(1x,A)")  "beams momentum: [undefined]"
      end select
    end subroutine cmd_beams_momentum_write
```

Compile: inherited.

Command execution: Not inherited, but just the error string and the final command are changed.

⟨*Commands: cmd beams momentum: TBP*⟩+≡
```
    procedure :: execute => cmd_beams_momentum_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_beams_momentum_execute (cmd, global)
      class(cmd_beams_momentum_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      real(default), dimension(:), allocatable :: p
      type(eval_tree_t) :: expr
      integer :: i
      allocate (p (cmd%n_in))
      do i = 1, cmd%n_in
         call eval_tree_init_expr (expr, cmd%expr(i)%ptr, global%var_list)
         call eval_tree_evaluate (expr)
         if (eval_tree_result_is_known (expr)) then
            p(i) = eval_tree_get_real (expr)
         else
            call msg_fatal ("beams momentum: undefined value")
         end if
         call eval_tree_final (expr)
      end do
      call global%beam_structure%set_momentum (p)
    end subroutine cmd_beams_momentum_execute
```

Beam angles

Again, this is analogous. There are two angles, polar angle θ and azimuthal angle ϕ, which can be set independently for both beams.

⟨*Commands: types*⟩+≡
```
type, extends (cmd_beams_pol_fraction_t) :: cmd_beams_theta_t
 contains
 ⟨Commands: cmd beams theta: TBP⟩
end type cmd_beams_theta_t

type, extends (cmd_beams_pol_fraction_t) :: cmd_beams_phi_t
 contains
 ⟨Commands: cmd beams phi: TBP⟩
end type cmd_beams_phi_t
```

Output.

⟨*Commands: cmd beams theta: TBP*⟩≡
```
procedure :: write => cmd_beams_theta_write
```

⟨*Commands: cmd beams phi: TBP*⟩≡
```
procedure :: write => cmd_beams_phi_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_beams_theta_write (cmd, unit, indent)
  class(cmd_beams_theta_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  select case (cmd%n_in)
  case (1)
     write (u, "(1x,A)")  "beams theta: 1 [decay]"
  case (2)
     write (u, "(1x,A)")  "beams theta: 2 [scattering]"
  case default
     write (u, "(1x,A)")  "beams theta: [undefined]"
  end select
end subroutine cmd_beams_theta_write

subroutine cmd_beams_phi_write (cmd, unit, indent)
  class(cmd_beams_phi_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  select case (cmd%n_in)
  case (1)
     write (u, "(1x,A)")  "beams phi: 1 [decay]"
  case (2)
     write (u, "(1x,A)")  "beams phi: 2 [scattering]"
  case default
     write (u, "(1x,A)")  "beams phi: [undefined]"
  end select
end subroutine cmd_beams_phi_write
```

Compile: inherited.

Command execution: Not inherited, but just the error string and the final command are changed.

⟨*Commands: cmd beams theta: TBP*⟩+≡

```
procedure :: execute => cmd_beams_theta_execute
```

⟨*Commands: cmd beams phi: TBP*⟩+≡

```
procedure :: execute => cmd_beams_phi_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_theta_execute (cmd, global)
  class(cmd_beams_theta_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  real(default), dimension(:), allocatable :: theta
  type(eval_tree_t) :: expr
  integer :: i
  allocate (theta (cmd%n_in))
  do i = 1, cmd%n_in
     call eval_tree_init_expr (expr, cmd%expr(i)%ptr, global%var_list)
     call eval_tree_evaluate (expr)
     if (eval_tree_result_is_known (expr)) then
        theta(i) = eval_tree_get_real (expr)
     else
        call msg_fatal ("beams theta: undefined value")
     end if
     call eval_tree_final (expr)
  end do
  call global%beam_structure%set_theta (theta)
end subroutine cmd_beams_theta_execute

subroutine cmd_beams_phi_execute (cmd, global)
  class(cmd_beams_phi_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  real(default), dimension(:), allocatable :: phi
  type(eval_tree_t) :: expr
  integer :: i
  allocate (phi (cmd%n_in))
  do i = 1, cmd%n_in
     call eval_tree_init_expr (expr, cmd%expr(i)%ptr, global%var_list)
     call eval_tree_evaluate (expr)
     if (eval_tree_result_is_known (expr)) then
        phi(i) = eval_tree_get_real (expr)
     else
        call msg_fatal ("beams phi: undefined value")
     end if
     call eval_tree_final (expr)
  end do
  call global%beam_structure%set_phi (phi)
end subroutine cmd_beams_phi_execute
```

Cuts

Define a cut expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the

cut expression is used.

⟨*Commands: types*⟩+≡

```
    type, extends (command_t) :: cmd_cuts_t
      private
      type(parse_node_t), pointer :: pn_lexpr => null ()
     contains
     ⟨Commands: cmd cuts: TBP⟩
    end type cmd_cuts_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that cuts have been defined.

⟨*Commands: cmd cuts: TBP*⟩≡

```
    procedure :: write => cmd_cuts_write
```

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_cuts_write (cmd, unit, indent)
      class(cmd_cuts_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "cuts: [defined]"
    end subroutine cmd_cuts_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd cuts: TBP*⟩+≡

```
    procedure :: compile => cmd_cuts_compile
```

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_cuts_compile (cmd, global)
      class(cmd_cuts_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_cuts_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd cuts: TBP*⟩+≡

```
    procedure :: execute => cmd_cuts_execute
```

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_cuts_execute (cmd, global)
      class(cmd_cuts_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%cuts_lexpr => cmd%pn_lexpr
    end subroutine cmd_cuts_execute
```

General, Factorization and Renormalization Scales

Define a scale expression for either the renormalization or the factorization scale. We store the parse tree for the right-hand side instead of compiling

it. Compilation is deferred to the process environment where the expression is used.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_scale_t
      private
      type(parse_node_t), pointer :: pn_expr => null ()
     contains
     ⟨Commands: cmd scale: TBP⟩
    end type cmd_scale_t
```

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_fac_scale_t
      private
      type(parse_node_t), pointer :: pn_expr => null ()
     contains
     ⟨Commands: cmd fac scale: TBP⟩
    end type cmd_fac_scale_t
```

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_ren_scale_t
      private
      type(parse_node_t), pointer :: pn_expr => null ()
     contains
     ⟨Commands: cmd ren scale: TBP⟩
    end type cmd_ren_scale_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd scale: TBP*⟩≡
```
    procedure :: write => cmd_scale_write
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_scale_write (cmd, unit, indent)
      class(cmd_scale_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "scale: [defined]"
    end subroutine cmd_scale_write
```

⟨*Commands: cmd fac scale: TBP*⟩≡
```
    procedure :: write => cmd_fac_scale_write
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_fac_scale_write (cmd, unit, indent)
      class(cmd_fac_scale_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "factorization scale: [defined]"
    end subroutine cmd_fac_scale_write
```

⟨*Commands: cmd ren scale: TBP*⟩≡
```
    procedure :: write => cmd_ren_scale_write
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_ren_scale_write (cmd, unit, indent)
      class(cmd_ren_scale_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "renormalization scale: [defined]"
    end subroutine cmd_ren_scale_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd scale: TBP*⟩+≡
```
    procedure :: compile => cmd_scale_compile
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_scale_compile (cmd, global)
      class(cmd_scale_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_scale_compile
```

⟨*Commands: cmd fac scale: TBP*⟩+≡
```
    procedure :: compile => cmd_fac_scale_compile
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_fac_scale_compile (cmd, global)
      class(cmd_fac_scale_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_fac_scale_compile
```

⟨*Commands: cmd ren scale: TBP*⟩+≡
```
    procedure :: compile => cmd_ren_scale_compile
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_ren_scale_compile (cmd, global)
      class(cmd_ren_scale_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_ren_scale_compile
```

Instead of evaluating the scale expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd scale: TBP*⟩+≡
```
    procedure :: execute => cmd_scale_execute
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_scale_execute (cmd, global)
      class(cmd_scale_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%scale_expr => cmd%pn_expr
    end subroutine cmd_scale_execute
```

⟨*Commands: cmd fac scale: TBP*⟩+≡
```
    procedure :: execute => cmd_fac_scale_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_fac_scale_execute (cmd, global)
      class(cmd_fac_scale_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%fac_scale_expr => cmd%pn_expr
    end subroutine cmd_fac_scale_execute
```

⟨*Commands: cmd ren scale: TBP*⟩+≡
```
    procedure :: execute => cmd_ren_scale_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_ren_scale_execute (cmd, global)
      class(cmd_ren_scale_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%ren_scale_expr => cmd%pn_expr
    end subroutine cmd_ren_scale_execute
```

Weight

Define a weight expression. The weight is applied to a process to be integrated, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the expression is used.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_weight_t
      private
      type(parse_node_t), pointer :: pn_expr => null ()
     contains
     ⟨Commands: cmd weight: TBP⟩
    end type cmd_weight_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd weight: TBP*⟩≡
```
    procedure :: write => cmd_weight_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_weight_write (cmd, unit, indent)
      class(cmd_weight_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "weight expression: [defined]"
    end subroutine cmd_weight_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd weight: TBP*⟩+≡
```
    procedure :: compile => cmd_weight_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_weight_compile (cmd, global)
      class(cmd_weight_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_weight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set,
such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd weight: TBP*⟩+≡
```
    procedure :: execute => cmd_weight_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_weight_execute (cmd, global)
      class(cmd_weight_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%weight_expr => cmd%pn_expr
    end subroutine cmd_weight_execute
```

Selection

Define a selection expression. This is to be applied upon simulation or event-
file rescanning, event by event. We store the parse tree for the right-hand side
instead of compiling it. Compilation is deferred to the environment where the
expression is used.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_selection_t
      private
      type(parse_node_t), pointer :: pn_expr => null ()
     contains
     ⟨Commands: cmd selection: TBP⟩
    end type cmd_selection_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message
that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd selection: TBP*⟩≡
```
    procedure :: write => cmd_selection_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_selection_write (cmd, unit, indent)
      class(cmd_selection_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "selection expression: [defined]"
    end subroutine cmd_selection_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd selection: TBP*⟩+≡
```
    procedure :: compile => cmd_selection_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_selection_compile (cmd, global)
      class(cmd_selection_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_selection_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd selection: TBP*⟩+≡
```
    procedure :: execute => cmd_selection_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_selection_execute (cmd, global)
      class(cmd_selection_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%selection_lexpr => cmd%pn_expr
    end subroutine cmd_selection_execute
```

Reweight

Define a reweight expression. This is to be applied upon simulation or event-file rescanning, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the environment where the expression is used.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_reweight_t
      private
      type(parse_node_t), pointer :: pn_expr => null ()
     contains
     ⟨Commands: cmd reweight: TBP⟩
    end type cmd_reweight_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd reweight: TBP*⟩≡
```
    procedure :: write => cmd_reweight_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_reweight_write (cmd, unit, indent)
      class(cmd_reweight_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "reweight expression: [defined]"
    end subroutine cmd_reweight_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd reweight: TBP*⟩+≡
```
    procedure :: compile => cmd_reweight_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_reweight_compile (cmd, global)
      class(cmd_reweight_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_reweight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd reweight: TBP*⟩+≡
```
    procedure :: execute => cmd_reweight_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_reweight_execute (cmd, global)
      class(cmd_reweight_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%reweight_expr => cmd%pn_expr
    end subroutine cmd_reweight_execute
```

Alternative Simulation Setups

Together with simulation, we can re-evaluate event weights in the context of alternative setups. The `cmd_alt_setup_t` object is designed to hold these setups, which are brace-enclosed command lists. Compilation is deferred to the simulation environment where the setup expression is used.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_alt_setup_t
      private
      type(parse_node_p), dimension(:), allocatable :: setup
    contains
    ⟨Commands: cmd alt setup: TBP⟩
    end type cmd_alt_setup_t
```

Output. Print just a message that the alternative setup list has been defined.

⟨*Commands: cmd alt setup: TBP*⟩≡
```
    procedure :: write => cmd_alt_setup_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_alt_setup_write (cmd, unit, indent)
      class(cmd_alt_setup_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,I0,A)")  "alt_setup: ", size (cmd%setup), " entries"
    end subroutine cmd_alt_setup_write
```

Compile. Store the parse sub-trees in an array.

⟨*Commands: cmd alt setup: TBP*⟩+≡
```
    procedure :: compile => cmd_alt_setup_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_alt_setup_compile (cmd, global)
      class(cmd_alt_setup_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_list, pn_setup
      integer :: i
      pn_list => parse_node_get_sub_ptr (cmd%pn, 3)
      if (associated (pn_list)) then
         allocate (cmd%setup (parse_node_get_n_sub (pn_list)))
         i = 1
         pn_setup => parse_node_get_sub_ptr (pn_list)
         do while (associated (pn_setup))
            cmd%setup(i)%ptr => pn_setup
            i = i + 1
            pn_setup => parse_node_get_next_ptr (pn_setup)
         end do
      else
         allocate (cmd%setup (0))
      end if
    end subroutine cmd_alt_setup_compile
```

Execute. Transfer the array of command lists to the global environment.

⟨*Commands: cmd alt setup: TBP*⟩+≡
```
    procedure :: execute => cmd_alt_setup_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_alt_setup_execute (cmd, global)
      class(cmd_alt_setup_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      if (allocated (global%pn%alt_setup))  deallocate (global%pn%alt_setup)
      allocate (global%pn%alt_setup (size (cmd%setup)), source = cmd%setup)
    end subroutine cmd_alt_setup_execute
```

Integration

Integrate several processes, consecutively with identical parameters.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_integrate_t
      private
      integer :: n_proc = 0
      type(string_t), dimension(:), allocatable :: process_id
     contains
     ⟨Commands: cmd integrate: TBP⟩
    end type cmd_integrate_t
```

Output: we know the process IDs.

⟨*Commands: cmd integrate: TBP*⟩≡
```
    procedure :: write => cmd_integrate_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_integrate_write (cmd, unit, indent)
      class(cmd_integrate_t), intent(in) :: cmd
```

```
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "integrate ("
      do i = 1, cmd%n_proc
         if (i > 1)  write (u, "(A,1x)", advance="no")  ","
         write (u, "(A)", advance="no")  char (cmd%process_id(i))
      end do
      write (u, "(A)")  ")"
    end subroutine cmd_integrate_write
```

Compile.

⟨*Commands: cmd integrate: TBP*⟩+≡
```
    procedure :: compile => cmd_integrate_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_integrate_compile (cmd, global)
      class(cmd_integrate_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_proclist, pn_proc
      integer :: i
      pn_proclist => parse_node_get_sub_ptr (cmd%pn, 2)
      cmd%pn_opt => parse_node_get_next_ptr (pn_proclist)
      call cmd%compile_options (global)
      cmd%n_proc = parse_node_get_n_sub (pn_proclist)
      allocate (cmd%process_id (cmd%n_proc))
      pn_proc => parse_node_get_sub_ptr (pn_proclist)
      do i = 1, cmd%n_proc
         cmd%process_id(i) = parse_node_get_string (pn_proc)
         call global%process_stack%init_result_vars (cmd%process_id(i))
         pn_proc => parse_node_get_next_ptr (pn_proc)
      end do
    end subroutine cmd_integrate_compile
```

Command execution. Integrate the process(es) with the predefined number of passes, iterations and calls. For structure functions, cuts, weight and scale, use local definitions if present; by default, the local definitions are initialized with the global ones.

Since the process acquires a snapshot of the variable list, so if the global list (or the local one) is deleted, this does no harm. This implies that later changes of the variable list do not affect the stored process.

⟨*Commands: cmd integrate: TBP*⟩+≡
```
    procedure :: execute => cmd_integrate_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_integrate_execute (cmd, global)
      class(cmd_integrate_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      integer :: i
      do i = 1, cmd%n_proc
         call integrate_process (cmd%process_id(i), cmd%local)
         call cmd%local%process_stack%fill_result_vars (cmd%process_id(i))
```

```
        if (signal_is_pending ())  return
      end do
    end subroutine cmd_integrate_execute
```

Observables

Declare an observable. After the declaration, it can be used to record data, and at the end one can retrieve average and error.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_observable_t
      private
      type(string_t) :: id
    contains
    ⟨Commands: cmd observable: TBP⟩
    end type cmd_observable_t
```

Output. We know the ID.

⟨*Commands: cmd observable: TBP*⟩≡
```
    procedure :: write => cmd_observable_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_observable_write (cmd, unit, indent)
      class(cmd_observable_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,A)")  "observable: ", char (cmd%id)
    end subroutine cmd_observable_write
```

Compile. Just record the observable ID.

⟨*Commands: cmd observable: TBP*⟩+≡
```
    procedure :: compile => cmd_observable_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_observable_compile (cmd, global)
      class(cmd_observable_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_tag
      pn_tag => parse_node_get_sub_ptr (cmd%pn, 2)
      if (associated (pn_tag)) then
         cmd%pn_opt => parse_node_get_next_ptr (pn_tag)
      end if
      call cmd%compile_options (global)
      select case (char (parse_node_get_rule_key (pn_tag)))
      case ("analysis_id")
         cmd%id = parse_node_get_string (pn_tag)
      case default
         call msg_bug ("observable: name expression not implemented (yet)")
      end select
    end subroutine cmd_observable_compile
```

Command execution. This declares the observable and allocates it in the analysis store.

⟨*Commands: cmd observable: TBP*⟩+≡
```
    procedure :: execute => cmd_observable_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_observable_execute (cmd, global)
      class(cmd_observable_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(graph_options_t) :: graph_options
      type(string_t) :: label, unit
      label = var_list_get_sval &
           (cmd%local%var_list, var_str ("$obs_label"))
      unit = var_list_get_sval &
           (cmd%local%var_list, var_str ("$obs_unit"))
      call graph_options_init (graph_options)
      call set_graph_options (graph_options, cmd%local%var_list)
      call analysis_init_observable (cmd%id, label, unit, graph_options)
    end subroutine cmd_observable_execute
```

Histograms

Declare a histogram. At minimum, we have to set lower and upper bound and bin width.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_histogram_t
      private
      type(string_t) :: id
      type(parse_node_t), pointer :: pn_lower_bound => null ()
      type(parse_node_t), pointer :: pn_upper_bound => null ()
      type(parse_node_t), pointer :: pn_bin_width => null ()
    contains
    ⟨Commands: cmd histogram: TBP⟩
    end type cmd_histogram_t
```

Output. Just print the ID.

⟨*Commands: cmd histogram: TBP*⟩≡
```
    procedure :: write => cmd_histogram_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_histogram_write (cmd, unit, indent)
      class(cmd_histogram_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,A)")  "histogram: ", char (cmd%id)
    end subroutine cmd_histogram_write
```

Compile. Record the histogram ID and initialize lower, upper bound and bin width.

⟨*Commands: cmd histogram: TBP*⟩+≡
```
    procedure :: compile => cmd_histogram_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_histogram_compile (cmd, global)
      class(cmd_histogram_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_tag, pn_args, pn_arg1, pn_arg2, pn_arg3
      character(*), parameter :: e_illegal_use = &
         "illegal usage of 'histogram': insufficient number of arguments"
      pn_tag => parse_node_get_sub_ptr (cmd%pn, 2)
      pn_args => parse_node_get_next_ptr (pn_tag)
      if (associated (pn_args)) then
         pn_arg1 => parse_node_get_sub_ptr (pn_args)
         if (.not. associated (pn_arg1)) call msg_fatal (e_illegal_use)
         pn_arg2 => parse_node_get_next_ptr (pn_arg1)
         if (.not. associated (pn_arg2)) call msg_fatal (e_illegal_use)
         pn_arg3 => parse_node_get_next_ptr (pn_arg2)
         cmd%pn_opt => parse_node_get_next_ptr (pn_args)
      end if
      call cmd%compile_options (global)
      select case (char (parse_node_get_rule_key (pn_tag)))
      case ("analysis_id")
         cmd%id = parse_node_get_string (pn_tag)
      case default
         call msg_bug ("histogram: name expression not implemented (yet)")
      end select
      cmd%pn_lower_bound => pn_arg1
      cmd%pn_upper_bound => pn_arg2
      cmd%pn_bin_width => pn_arg3
    end subroutine cmd_histogram_compile
```

Command execution. This declares the histogram and allocates it in the analysis store.

⟨*Commands: cmd histogram: TBP*⟩+≡
```
    procedure :: execute => cmd_histogram_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_histogram_execute (cmd, global)
      class(cmd_histogram_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      real(default) :: lower_bound, upper_bound, bin_width
      integer :: bin_number
      logical :: bin_width_is_used, normalize_bins
      type(string_t) :: obs_label, obs_unit
      type(graph_options_t) :: graph_options
      type(drawing_options_t) :: drawing_options

      lower_bound = eval_real (cmd%pn_lower_bound, cmd%local%var_list)
      upper_bound = eval_real (cmd%pn_upper_bound, cmd%local%var_list)
      if (associated (cmd%pn_bin_width)) then
         bin_width = eval_real (cmd%pn_bin_width, cmd%local%var_list)
         bin_width_is_used = .true.
      else if (var_list_is_known &
         (cmd%local%var_list, var_str ("n_bins"))) then
         bin_number = var_list_get_ival &
            (cmd%local%var_list, var_str ("n_bins"))
```

```
      bin_width_is_used = .false.
   else
      call msg_error ("Cmd '" // char (cmd%id) // &
            "': neither bin width nor number is defined")
   end if
   normalize_bins = var_list_get_lval &
         (cmd%local%var_list, var_str ("?normalize_bins"))
   obs_label = var_list_get_sval &
         (cmd%local%var_list, var_str ("$obs_label"))
   obs_unit = var_list_get_sval &
         (cmd%local%var_list, var_str ("$obs_unit"))

   call graph_options_init (graph_options)
   call set_graph_options (graph_options, cmd%local%var_list)
   call drawing_options_init_histogram (drawing_options)
   call set_drawing_options (drawing_options, cmd%local%var_list)

   if (bin_width_is_used) then
      call analysis_init_histogram &
            (cmd%id, lower_bound, upper_bound, bin_width, &
             normalize_bins, &
             obs_label, obs_unit, &
             graph_options, drawing_options)
   else
      call analysis_init_histogram &
            (cmd%id, lower_bound, upper_bound, bin_number, &
             normalize_bins, &
             obs_label, obs_unit, &
             graph_options, drawing_options)
   end if
end subroutine cmd_histogram_execute
```

Set the graph options from a variable list.
⟨*Commands: procedures*⟩+≡
```
   subroutine set_graph_options (gro, var_list)
     type(graph_options_t), intent(inout) :: gro
     type(var_list_t), intent(in) :: var_list
     call graph_options_set (gro, title = &
          var_list_get_sval (var_list, var_str ("$title")))
     call graph_options_set (gro, description = &
          var_list_get_sval (var_list, var_str ("$description")))
     call graph_options_set (gro, x_label = &
          var_list_get_sval (var_list, var_str ("$x_label")))
     call graph_options_set (gro, y_label = &
          var_list_get_sval (var_list, var_str ("$y_label")))
     call graph_options_set (gro, width_mm = &
          var_list_get_ival (var_list, var_str ("graph_width_mm")))
     call graph_options_set (gro, height_mm = &
          var_list_get_ival (var_list, var_str ("graph_height_mm")))
     call graph_options_set (gro, x_log = &
          var_list_get_lval (var_list, var_str ("?x_log")))
     call graph_options_set (gro, y_log = &
          var_list_get_lval (var_list, var_str ("?y_log")))
```

```
      if (var_list_is_known (var_list, var_str ("x_min"))) &
           call graph_options_set (gro, x_min = &
           var_list_get_rval (var_list, var_str ("x_min")))
      if (var_list_is_known (var_list, var_str ("x_max"))) &
           call graph_options_set (gro, x_max = &
           var_list_get_rval (var_list, var_str ("x_max")))
      if (var_list_is_known (var_list, var_str ("y_min"))) &
           call graph_options_set (gro, y_min = &
           var_list_get_rval (var_list, var_str ("y_min")))
      if (var_list_is_known (var_list, var_str ("y_max"))) &
           call graph_options_set (gro, y_max = &
           var_list_get_rval (var_list, var_str ("y_max")))
      call graph_options_set (gro, gmlcode_bg = &
           var_list_get_sval (var_list, var_str ("$gmlcode_bg")))
      call graph_options_set (gro, gmlcode_fg = &
           var_list_get_sval (var_list, var_str ("$gmlcode_fg")))
    end subroutine set_graph_options
```

Set the drawing options from a variable list.
⟨*Commands: procedures*⟩+≡
```
    subroutine set_drawing_options (dro, var_list)
      type(drawing_options_t), intent(inout) :: dro
      type(var_list_t), intent(in) :: var_list
      if (var_list_is_known (var_list, var_str ("?draw_histogram"))) then
         if (var_list_get_lval (var_list, var_str ("?draw_histogram"))) then
            call drawing_options_set (dro, with_hbars = .true.)
         else
            call drawing_options_set (dro, with_hbars = .false., &
                  with_base = .false., fill = .false., piecewise = .false.)
         end if
      end if
      if (var_list_is_known (var_list, var_str ("?draw_base"))) then
         if (var_list_get_lval (var_list, var_str ("?draw_base"))) then
            call drawing_options_set (dro, with_base = .true.)
         else
            call drawing_options_set (dro, with_base = .false., fill = .false.)
         end if
      end if
      if (var_list_is_known (var_list, var_str ("?draw_piecewise"))) then
         if (var_list_get_lval (var_list, var_str ("?draw_piecewise"))) then
            call drawing_options_set (dro, piecewise = .true.)
         else
            call drawing_options_set (dro, piecewise = .false.)
         end if
      end if
      if (var_list_is_known (var_list, var_str ("?fill_curve"))) then
         if (var_list_get_lval (var_list, var_str ("?fill_curve"))) then
            call drawing_options_set (dro, fill = .true., with_base = .true.)
         else
            call drawing_options_set (dro, fill = .false.)
         end if
      end if
      if (var_list_is_known (var_list, var_str ("?draw_curve"))) then
         if (var_list_get_lval (var_list, var_str ("?draw_curve"))) then
```

```
            call drawing_options_set (dro, draw = .true.)
        else
            call drawing_options_set (dro, draw = .false.)
        end if
    end if
    if (var_list_is_known (var_list, var_str ("?draw_errors"))) then
        if (var_list_get_lval (var_list, var_str ("?draw_errors"))) then
            call drawing_options_set (dro, err = .true.)
        else
            call drawing_options_set (dro, err = .false.)
        end if
    end if
    if (var_list_is_known (var_list, var_str ("?draw_symbols"))) then
        if (var_list_get_lval (var_list, var_str ("?draw_symbols"))) then
            call drawing_options_set (dro, symbols = .true.)
        else
            call drawing_options_set (dro, symbols = .false.)
        end if
    end if
    if (var_list_is_known (var_list, var_str ("$fill_options"))) then
        call drawing_options_set (dro, fill_options = &
            var_list_get_sval (var_list, var_str ("$fill_options")))
    end if
    if (var_list_is_known (var_list, var_str ("$draw_options"))) then
        call drawing_options_set (dro, draw_options = &
            var_list_get_sval (var_list, var_str ("$draw_options")))
    end if
    if (var_list_is_known (var_list, var_str ("$err_options"))) then
        call drawing_options_set (dro, err_options = &
            var_list_get_sval (var_list, var_str ("$err_options")))
    end if
    if (var_list_is_known (var_list, var_str ("$symbol"))) then
        call drawing_options_set (dro, symbol = &
            var_list_get_sval (var_list, var_str ("$symbol")))
    end if
    if (var_list_is_known (var_list, var_str ("$gmlcode_bg"))) then
        call drawing_options_set (dro, gmlcode_bg = &
            var_list_get_sval (var_list, var_str ("$gmlcode_bg")))
    end if
    if (var_list_is_known (var_list, var_str ("$gmlcode_fg"))) then
        call drawing_options_set (dro, gmlcode_fg = &
            var_list_get_sval (var_list, var_str ("$gmlcode_fg")))
    end if
  end subroutine set_drawing_options
```

Plots

Declare a plot. No mandatory arguments, just options.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_plot_t
      private
      type(string_t) :: id
    contains
```

⟨*Commands: cmd plot: TBP*⟩
```
    end type cmd_plot_t
```

Output. Just print the ID.

⟨*Commands: cmd plot: TBP*⟩≡
```
    procedure :: write => cmd_plot_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_plot_write (cmd, unit, indent)
      class(cmd_plot_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,A)")  "plot: ", char (cmd%id)
    end subroutine cmd_plot_write
```

Compile. Record the plot ID and initialize lower, upper bound and bin width.

⟨*Commands: cmd plot: TBP*⟩+≡
```
    procedure :: compile => cmd_plot_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_plot_compile (cmd, global)
      class(cmd_plot_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_tag
      pn_tag => parse_node_get_sub_ptr (cmd%pn, 2)
      cmd%pn_opt => parse_node_get_next_ptr (pn_tag)
      call cmd%init (pn_tag, global)
    end subroutine cmd_plot_compile
```

This init routine is separated because it is reused below for graph initialization.

⟨*Commands: cmd plot: TBP*⟩+≡
```
    procedure :: init => cmd_plot_init
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_plot_init (plot, pn_tag, global)
      class(cmd_plot_t), intent(inout) :: plot
      type(parse_node_t), intent(in), pointer :: pn_tag
      type(rt_data_t), intent(inout), target :: global
      call plot%compile_options (global)
      select case (char (parse_node_get_rule_key (pn_tag)))
      case ("analysis_id")
         plot%id = parse_node_get_string (pn_tag)
      case default
         call msg_bug ("plot: name expression not implemented (yet)")
      end select
    end subroutine cmd_plot_init
```

Command execution. This declares the plot and allocates it in the analysis store.

⟨*Commands: cmd plot: TBP*⟩+≡
```
    procedure :: execute => cmd_plot_execute
```

3064

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_plot_execute (cmd, global)
      class(cmd_plot_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(graph_options_t) :: graph_options
      type(drawing_options_t) :: drawing_options

      call graph_options_init (graph_options)
      call set_graph_options (graph_options, cmd%local%var_list)
      call drawing_options_init_plot (drawing_options)
      call set_drawing_options (drawing_options, cmd%local%var_list)

      call analysis_init_plot (cmd%id, graph_options, drawing_options)
    end subroutine cmd_plot_execute
```

Graphs

Declare a graph. The graph is defined in terms of its contents. Both the graph
and its contents may carry options.

The graph object contains its own ID as well as the IDs of its elements. For
the elements, we reuse the `cmd_plot_t` defined above.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_graph_t
      private
      type(string_t) :: id
      integer :: n_elements = 0
      type(cmd_plot_t), dimension(:), allocatable :: el
      type(string_t), dimension(:), allocatable :: element_id
    contains
    ⟨Commands: cmd graph: TBP⟩
    end type cmd_graph_t
```

Output. Just print the ID.

⟨*Commands: cmd graph: TBP*⟩≡
```
    procedure :: write => cmd_graph_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_graph_write (cmd, unit, indent)
      class(cmd_graph_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,A,A,I0,A)")  "graph: ", char (cmd%id), &
           " (", cmd%n_elements, " entries)"
    end subroutine cmd_graph_write
```

Compile. Record the graph ID and initialize lower, upper bound and bin width.
For compiling the graph element syntax, we use part of the `cmd_plot_t` com-
piler.

Note: currently, we do not respect options, therefore just IDs on the RHS.

⟨*Commands: cmd graph: TBP*⟩+≡
```
    procedure :: compile => cmd_graph_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_graph_compile (cmd, global)
      class(cmd_graph_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_term, pn_tag, pn_def, pn_app
      integer :: i

      pn_term => parse_node_get_sub_ptr (cmd%pn, 2)
      pn_tag => parse_node_get_sub_ptr (pn_term)
      cmd%pn_opt => parse_node_get_next_ptr (pn_tag)
      call cmd%compile_options (global)
      select case (char (parse_node_get_rule_key (pn_tag)))
      case ("analysis_id")
         cmd%id = parse_node_get_string (pn_tag)
      case default
         call msg_bug ("graph: name expression not implemented (yet)")
      end select
      pn_def => parse_node_get_next_ptr (pn_term, 2)
      cmd%n_elements = parse_node_get_n_sub (pn_def)
      allocate (cmd%element_id (cmd%n_elements))
      allocate (cmd%el (cmd%n_elements))
      pn_term => parse_node_get_sub_ptr (pn_def)
      pn_tag => parse_node_get_sub_ptr (pn_term)
      cmd%el(1)%pn_opt => parse_node_get_next_ptr (pn_tag)
      call cmd%el(1)%init (pn_tag, global)
      cmd%element_id(1) = parse_node_get_string (pn_tag)
      pn_app => parse_node_get_next_ptr (pn_term)
      do i = 2, cmd%n_elements
         pn_term => parse_node_get_sub_ptr (pn_app, 2)
         pn_tag => parse_node_get_sub_ptr (pn_term)
         cmd%el(i)%pn_opt => parse_node_get_next_ptr (pn_tag)
         call cmd%el(i)%init (pn_tag, global)
         cmd%element_id(i) = parse_node_get_string (pn_tag)
         pn_app => parse_node_get_next_ptr (pn_app)
      end do

    end subroutine cmd_graph_compile
```

Command execution. This declares the graph, allocates it in the analysis store, and copies the graph elements.

For the graph, we set graph and default drawing options. For the elements, we reset individual drawing options.

This accesses internals of the contained elements of type `cmd_plot_t`, see above. We might disentangle such an interdependency when this code is rewritten using proper type extension.

⟨*Commands: cmd graph: TBP*⟩+≡
```
    procedure :: execute => cmd_graph_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_graph_execute (cmd, global)
```

```
      class(cmd_graph_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(graph_options_t) :: graph_options
      type(drawing_options_t) :: drawing_options
      integer :: i, type

      call graph_options_init (graph_options)
      call set_graph_options (graph_options, cmd%local%var_list)
      call analysis_init_graph (cmd%id, cmd%n_elements, graph_options)

      do i = 1, cmd%n_elements
         if (associated (cmd%el(i)%options)) then
            call cmd%el(i)%options%execute (cmd%el(i)%local)
         end if
         type = analysis_store_get_object_type (cmd%element_id(i))
         select case (type)
         case (AN_HISTOGRAM)
            call drawing_options_init_histogram (drawing_options)
         case (AN_PLOT)
            call drawing_options_init_plot (drawing_options)
         end select
         call set_drawing_options (drawing_options, cmd%local%var_list)
         if (associated (cmd%el(i)%options)) then
            call set_drawing_options (drawing_options, cmd%el(i)%local%var_list)
         end if
         call analysis_fill_graph (cmd%id, i, cmd%element_id(i), drawing_options)
      end do
    end subroutine cmd_graph_execute
```

Analysis

Hold the analysis ID either as a string or as an expression:

⟨*Commands: types*⟩+≡
```
   type :: analysis_id_t
     type(string_t) :: tag
     type(parse_node_t), pointer :: pn_sexpr => null ()
   end type analysis_id_t
```

Define the analysis expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the analysis expression is used.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_analysis_t
     private
     type(parse_node_t), pointer :: pn_lexpr => null ()
   contains
   ⟨Commands: cmd analysis: TBP⟩
   end type cmd_analysis_t
```

Output. Print just a message that analysis has been defined.

⟨*Commands: cmd analysis: TBP*⟩≡

```
    procedure :: write => cmd_analysis_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_analysis_write (cmd, unit, indent)
      class(cmd_analysis_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "analysis: [defined]"
    end subroutine cmd_analysis_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd analysis: TBP*⟩+≡
```
    procedure :: compile => cmd_analysis_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_analysis_compile (cmd, global)
      class(cmd_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 3)
    end subroutine cmd_analysis_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd analysis: TBP*⟩+≡
```
    procedure :: execute => cmd_analysis_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_analysis_execute (cmd, global)
      class(cmd_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      global%pn%analysis_lexpr => cmd%pn_lexpr
    end subroutine cmd_analysis_execute
```

Write histograms and plots

The data type encapsulating the command:

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_write_analysis_t
      private
      type(analysis_id_t), dimension(:), allocatable :: id
      type(string_t), dimension(:), allocatable :: tag
     contains
     ⟨Commands: cmd write analysis: TBP⟩
    end type cmd_write_analysis_t
```

Output. Just the keyword.

⟨*Commands: cmd write analysis: TBP*⟩≡
```
    procedure :: write => cmd_write_analysis_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_write_analysis_write (cmd, unit, indent)
      class(cmd_write_analysis_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "write_analysis"
    end subroutine cmd_write_analysis_write
```

Compile.

⟨*Commands: cmd write analysis: TBP*⟩+≡
```
    procedure :: compile => cmd_write_analysis_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_write_analysis_compile (cmd, global)
      class(cmd_write_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_clause, pn_args, pn_id
      integer :: n, i
      pn_clause => parse_node_get_sub_ptr (cmd%pn)
      pn_args => parse_node_get_sub_ptr (pn_clause, 2)
      cmd%pn_opt => parse_node_get_next_ptr (pn_clause)
      call cmd%compile_options (global)
      if (associated (pn_args)) then
         n = parse_node_get_n_sub (pn_args)
         allocate (cmd%id (n))
         do i = 1, n
            pn_id => parse_node_get_sub_ptr (pn_args, i)
            if (char (parse_node_get_rule_key (pn_id)) == "analysis_id") then
               cmd%id(i)%tag = parse_node_get_string (pn_id)
            else
               cmd%id(i)%pn_sexpr => pn_id
            end if
         end do
      else
         allocate (cmd%id (0))
      end if
    end subroutine cmd_write_analysis_compile
```

The output format for real data values:

⟨*Limits: public parameters*⟩+≡
```
    character(*), parameter, public :: &
         DEFAULT_ANALYSIS_FILENAME = "whizard_analysis.dat"
    character(len=1), dimension(2), parameter, public :: &
         FORBIDDEN_ENDINGS1 = [ "o", "a" ]
    character(len=2), dimension(5), parameter, public :: &
         FORBIDDEN_ENDINGS2 = [ "mp", "ps", "vg", "lo", "la" ]
    character(len=3), dimension(14), parameter, public :: &
         FORBIDDEN_ENDINGS3 = [ "aux", "dvi", "evt", "evx", "f03", "f90", &
             "f95", "log", "ltp", "mpx", "pdf", "phs", "sin", "tex" ]
```

As this contains a lot of similar code to `cmd_compile_analysis_execute` we outsource the main code to a subroutine.

⟨*Commands: cmd write analysis: TBP*⟩+≡
```
    procedure :: execute => cmd_write_analysis_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_write_analysis_execute (cmd, global)
      class(cmd_write_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      call write_analysis_wrap (cmd%local%var_list, global%out_files, &
           cmd%id, tag = cmd%tag)
    end subroutine cmd_write_analysis_execute
```

If the `data_file` optional argument is present, this is called from `cmd_compile_analysis_execute`, which needs the file name for further processing, and requires the default format. For the moment, parameters and macros for custom data processing are disabled.

⟨*Commands: procedures*⟩+≡
```
    subroutine write_analysis_wrap (var_list, out_files, id, tag, data_file)
      type(var_list_t), intent(inout), target :: var_list
      type(file_list_t), intent(inout), target :: out_files
      type(analysis_id_t), dimension(:), intent(in), target :: id
      type(string_t), dimension(:), allocatable, intent(out) :: tag
      type(string_t), intent(out), optional :: data_file
      type(string_t) :: defaultfile, file
      integer :: i
      logical :: keep_open !, custom, header, columns
      type(string_t) :: extension !, comment_prefix, separator
!!! JRR: WK please check (#542)
!     integer :: type
!     type(ifile_t) :: ifile
      logical :: one_file !, has_writer
!     type(analysis_iterator_t) :: iterator
!     type(rt_data_t), target :: sandbox
!     type(command_list_t) :: writer
      defaultfile = var_list_get_sval (var_list, var_str ("$out_file"))
      if (present (data_file)) then
         if (defaultfile == "" .or. defaultfile == ".") then
            defaultfile = DEFAULT_ANALYSIS_FILENAME
         else
            if (scan (".", defaultfile) > 0) then
               call split (defaultfile, extension, ".", back=.true.)
               if (any (lower_case (char(extension)) == FORBIDDEN_ENDINGS1) .or. &
                   any (lower_case (char(extension)) == FORBIDDEN_ENDINGS2) .or. &
                   any (lower_case (char(extension)) == FORBIDDEN_ENDINGS3)) &
                   call msg_fatal ("The ending " // char(extension) // &
                   " is internal and not allowed as data file.")
               if (extension /= "") then
                  if (defaultfile /= "") then
                     defaultfile = defaultfile // "." // extension
                  else
                     defaultfile = "whizard_analysis." // extension
                  end if
```

3070

```
              else
                 defaultfile = defaultfile // ".dat"
              endif
           else
              defaultfile = defaultfile // ".dat"
           end if
        end if
        data_file = defaultfile
     end if
     one_file = defaultfile /= ""
     if (one_file) then
        file = defaultfile
        keep_open = file_list_is_open (out_files, file, &
              action = "write")
        if (keep_open) then
           if (present (data_file)) then
              call msg_fatal ("Compiling analysis: File '" &
                    // char (data_file) &
                     // "' can't be used, it is already open.")
           else
              call msg_message ("Appending analysis data to file '" &
                    // char (file) // "'")
           end if
        else
           call file_list_open (out_files, file, &
                 action = "write", status = "replace", position = "asis")
           call msg_message ("Writing analysis data to file '" &
                 // char (file) // "'")
        end if
     end if

!!! JRR: WK please check. Custom data output. Ticket #542
!     if (present (data_file)) then
!        custom = .false.
!     else
!        custom = var_list_get_lval (var_list, &
!            var_str ("?out_custom"))
!     end if
!     comment_prefix = var_list_get_sval (var_list, &
!          var_str ("$out_comment"))
!     header = var_list_get_lval (var_list, &
!          var_str ("?out_header"))
!     write_yerr = var_list_get_lval (var_list, &
!          var_str ("?out_yerr"))
!     write_xerr = var_list_get_lval (var_list, &
!          var_str ("?out_xerr"))

     call get_analysis_tags (tag, id, var_list)
     do i = 1, size (tag)
        call file_list_write_analysis &
              (out_files, file, tag(i))
     end do
     if (one_file .and. .not. keep_open) then
        call file_list_close (out_files, file)
```

```
      end if

   contains

      subroutine get_analysis_tags (analysis_tag, id, var_list)
         type(string_t), dimension(:), intent(out), allocatable :: analysis_tag
         type(analysis_id_t), dimension(:), intent(in) :: id
         type(var_list_t), intent(in), target :: var_list
         if (size (id) /= 0) then
            allocate (analysis_tag (size (id)))
            do i = 1, size (id)
               if (associated (id(i)%pn_sexpr)) then
                  analysis_tag(i) = eval_string (id(i)%pn_sexpr, var_list)
               else
                  analysis_tag(i) = id(i)%tag
               end if
            end do
         else
            call analysis_store_get_ids (tag)
         end if
      end subroutine get_analysis_tags

   end subroutine write_analysis_wrap
```

Compile analysis results

This command writes files in a form suitable for GAMELAN and executes
the appropriate commands to compile them. The first part is identical to
`cmd_write_analysis`.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_compile_analysis_t
      private
      type(analysis_id_t), dimension(:), allocatable :: id
      type(string_t), dimension(:), allocatable :: tag
    contains
    ⟨Commands: cmd compile analysis: TBP⟩
    end type cmd_compile_analysis_t
```

Output. Just the keyword.

⟨*Commands: cmd compile analysis: TBP*⟩≡
```
    procedure :: write => cmd_compile_analysis_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_compile_analysis_write (cmd, unit, indent)
      class(cmd_compile_analysis_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "compile_analysis"
    end subroutine cmd_compile_analysis_write
```

Compile.

⟨Commands: cmd compile analysis: TBP⟩+≡
```
    procedure :: compile => cmd_compile_analysis_compile
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_compile_analysis_compile (cmd, global)
      class(cmd_compile_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_clause, pn_args, pn_id
      integer :: n, i
      pn_clause => parse_node_get_sub_ptr (cmd%pn)
      pn_args => parse_node_get_sub_ptr (pn_clause, 2)
      cmd%pn_opt => parse_node_get_next_ptr (pn_clause)
      call cmd%compile_options (global)
      if (associated (pn_args)) then
         n = parse_node_get_n_sub (pn_args)
         allocate (cmd%id (n))
         do i = 1, n
            pn_id => parse_node_get_sub_ptr (pn_args, i)
            if (char (parse_node_get_rule_key (pn_id)) == "analysis_id") then
               cmd%id(i)%tag = parse_node_get_string (pn_id)
            else
               cmd%id(i)%pn_sexpr => pn_id
            end if
         end do
      else
         allocate (cmd%id (0))
      end if
    end subroutine cmd_compile_analysis_compile
```

First write the analysis data to file, then write a GAMELAN driver and produce
MetaPost and TEX output.

⟨Commands: cmd compile analysis: TBP⟩+≡
```
    procedure :: execute => cmd_compile_analysis_execute
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_compile_analysis_execute (cmd, global)
      class(cmd_compile_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(string_t) :: file, basename, extension, driver_file
      integer :: u_driver
      logical :: has_gmlcode
      call write_analysis_wrap (cmd%local%var_list, &
           global%out_files, cmd%id, tag = cmd%tag, &
              data_file = file)
      basename = file
      if (scan (".", basename) > 0) then
        call split (basename, extension, ".", back=.true.)
      else
        extension = ""
      end if
      driver_file = basename // ".tex"
      u_driver = free_unit ()
      open (unit=u_driver, file=char(driver_file), &
```

```
            action="write", status="replace")
      if (allocated (cmd%tag)) then
         call analysis_write_driver (file, cmd%tag, unit=u_driver)
         has_gmlcode = analysis_has_plots (cmd%tag)
      else
         call analysis_write_driver (file, unit=u_driver)
         has_gmlcode = analysis_has_plots ()
      end if
      close (u_driver)
      call msg_message ("Compiling analysis results display in '" &
           // char (driver_file) // "'")
      call analysis_compile_tex (basename, has_gmlcode, global%os_data)
   end subroutine cmd_compile_analysis_execute
```

23.2 User-controlled output to data files

Open file (output)

Open a file for output.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_open_out_t
      private
      type(parse_node_t), pointer :: file_expr => null ()
    contains
    ⟨Commands: cmd open out: TBP⟩
    end type cmd_open_out_t
```

Finalizer for the embedded eval tree.

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_open_out_final (object)
      class(cmd_open_out_t), intent(inout) :: object
    end subroutine cmd_open_out_final
```

Output (trivial here).

⟨*Commands: cmd open out: TBP*⟩≡
```
    procedure :: write => cmd_open_out_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_open_out_write (cmd, unit, indent)
      class(cmd_open_out_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "open_out: <filename>"
    end subroutine cmd_open_out_write
```

Compile: create an eval tree for the filename expression.

⟨*Commands: cmd open out: TBP*⟩+≡
```
    procedure :: compile => cmd_open_out_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_open_out_compile (cmd, global)
      class(cmd_open_out_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      cmd%file_expr => parse_node_get_sub_ptr (cmd%pn, 2)
      if (associated (cmd%file_expr)) then
          cmd%pn_opt => parse_node_get_next_ptr (cmd%file_expr)
      end if
      call cmd%compile_options (global)
    end subroutine cmd_open_out_compile
```

Execute: append the file to the global list of open files.

⟨*Commands: cmd open out: TBP*⟩+≡
```
    procedure :: execute => cmd_open_out_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_open_out_execute (cmd, global)
      class(cmd_open_out_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(eval_tree_t) :: file_expr
      type(string_t) :: file
      call eval_tree_init_sexpr (file_expr, cmd%file_expr, cmd%local%var_list)
      call eval_tree_evaluate (file_expr)
      if (eval_tree_result_is_known (file_expr)) then
          file = eval_tree_get_string (file_expr)
          call file_list_open (global%out_files, file, &
                action = "write", status = "replace", position = "asis")
      else
          call msg_fatal ("open_out: file name argument evaluates to unknown")
      end if
      call eval_tree_final (file_expr)
    end subroutine cmd_open_out_execute
```

Open file (output)

Close an output file. Except for the `execute` method, everything is analogous
to the open command, so we can just inherit.

⟨*Commands: types*⟩+≡
```
    type, extends (cmd_open_out_t) :: cmd_close_out_t
       private
     contains
     ⟨Commands: cmd close out: TBP⟩
    end type cmd_close_out_t
```

Execute: remove the file from the global list of output files.

⟨*Commands: cmd close out: TBP*⟩≡
```
    procedure :: execute => cmd_close_out_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_close_out_execute (cmd, global)
      class(cmd_close_out_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
```

```
    type(eval_tree_t) :: file_expr
    type(string_t) :: file
    call eval_tree_init_sexpr (file_expr, cmd%file_expr, cmd%local%var_list)
    call eval_tree_evaluate (file_expr)
    if (eval_tree_result_is_known (file_expr)) then
       file = eval_tree_get_string (file_expr)
       call file_list_close (global%out_files, file)
    else
       call msg_fatal ("close_out: file name argument evaluates to unknown")
    end if
    call eval_tree_final (file_expr)
end subroutine cmd_close_out_execute
```

23.2.1   Print custom-formatted values

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_printf_t
      private
      type(parse_node_t), pointer :: sexpr => null ()
      type(parse_node_t), pointer :: sprintf_fun => null ()
      type(parse_node_t), pointer :: sprintf_clause => null ()
      type(parse_node_t), pointer :: sprintf => null ()
     contains
     ⟨Commands: cmd printf: TBP⟩
    end type cmd_printf_t
```

Finalize.

⟨*Commands: cmd printf: TBP*⟩≡
```
    procedure :: final => cmd_printf_final
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_printf_final (cmd)
      class(cmd_printf_t), intent(inout) :: cmd
      call parse_node_final (cmd%sexpr, recursive = .false.)
      deallocate (cmd%sexpr)
      call parse_node_final (cmd%sprintf_fun, recursive = .false.)
      deallocate (cmd%sprintf_fun)
      call parse_node_final (cmd%sprintf_clause, recursive = .false.)
      deallocate (cmd%sprintf_clause)
      call parse_node_final (cmd%sprintf, recursive = .false.)
      deallocate (cmd%sprintf)
    end subroutine cmd_printf_final
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that cuts have been defined.

⟨*Commands: cmd printf: TBP*⟩+≡
```
    procedure :: write => cmd_printf_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_printf_write (cmd, unit, indent)
      class(cmd_printf_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
```

```
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)")  "printf:"
    end subroutine cmd_printf_write
```

Compile. We create a fake parse node (subtree) with a `sprintf` command with identical arguments which can then be handled by the corresponding evaluation procedure.

⟨*Commands: cmd printf: TBP*⟩+≡
```
    procedure :: compile => cmd_printf_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_printf_compile (cmd, global)
      class(cmd_printf_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_args, pn_format
      pn_cmd => parse_node_get_sub_ptr (cmd%pn)
      pn_clause => parse_node_get_sub_ptr (pn_cmd)
      pn_format => parse_node_get_sub_ptr (pn_clause, 2)
      pn_args => parse_node_get_next_ptr (pn_clause)
      cmd%pn_opt => parse_node_get_next_ptr (pn_cmd)
      call cmd%compile_options (global)
      allocate (cmd%sexpr)
      call parse_node_create_branch (cmd%sexpr, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("sexpr")))
      allocate (cmd%sprintf_fun)
      call parse_node_create_branch (cmd%sprintf_fun, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf_fun")))
      allocate (cmd%sprintf_clause)
      call parse_node_create_branch (cmd%sprintf_clause, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf_clause")))
      allocate (cmd%sprintf)
      call parse_node_create_key (cmd%sprintf, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf")))
      call parse_node_append_sub (cmd%sprintf_clause, cmd%sprintf)
      call parse_node_append_sub (cmd%sprintf_clause, pn_format)
      call parse_node_freeze_branch (cmd%sprintf_clause)
      call parse_node_append_sub (cmd%sprintf_fun, cmd%sprintf_clause)
      if (associated (pn_args)) then
         call parse_node_append_sub (cmd%sprintf_fun, pn_args)
      end if
      call parse_node_freeze_branch (cmd%sprintf_fun)
      call parse_node_append_sub (cmd%sexpr, cmd%sprintf_fun)
      call parse_node_freeze_branch (cmd%sexpr)
    end subroutine cmd_printf_compile
```

Execute. Evaluate the string (pretending this is a `sprintf` expression) and print it.

⟨*Commands: cmd printf: TBP*⟩+≡
```
    procedure :: execute => cmd_printf_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_printf_execute (cmd, global)
```

```
class(cmd_printf_t), intent(inout) :: cmd
type(rt_data_t), intent(inout), target :: global
type(string_t) :: string, file
type(eval_tree_t) :: sprintf_expr
logical :: advance
advance = var_list_get_lval (cmd%local%var_list, &
    var_str ("?out_advance"))
file = var_list_get_sval (cmd%local%var_list, &
    var_str ("$out_file"))
call eval_tree_init_sexpr (sprintf_expr, cmd%sexpr, cmd%local%var_list)
call eval_tree_evaluate (sprintf_expr)
if (eval_tree_result_is_known (sprintf_expr)) then
   string = eval_tree_get_string (sprintf_expr)
   if (len (file) == 0) then
      call msg_result (char (string))
   else
      call file_list_write (global%out_files, file, string, advance)
   end if
end if
end subroutine cmd_printf_execute
```

Record data

The expression syntax already contains a `record` keyword; this evaluates to a logical which is always true, but it has the side-effect of recording data into analysis objects. Here we define a command as an interface to this construct.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_record_t
  private
  type(parse_node_t), pointer :: pn_lexpr => null ()
 contains
 ⟨Commands: cmd record: TBP⟩
end type cmd_record_t
```

Output. With the compile hack below, there is nothing of interest to print here.

⟨*Commands: cmd record: TBP*⟩≡
```
procedure :: write => cmd_record_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_record_write (cmd, unit, indent)
  class(cmd_record_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "record"
end subroutine cmd_record_write
```

Compile. This is a hack which transforms the `record` command into a `record` expression, which we handle in the `expressions` module.

⟨*Commands: cmd record: TBP*⟩+≡
```
procedure :: compile => cmd_record_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_record_compile (cmd, global)
      class(cmd_record_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_lexpr, pn_lsinglet, pn_lterm, pn_record
      call parse_node_create_branch (pn_lexpr, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("lexpr")))
      call parse_node_create_branch (pn_lsinglet, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("lsinglet")))
      call parse_node_append_sub (pn_lexpr, pn_lsinglet)
      call parse_node_create_branch (pn_lterm, &
           syntax_get_rule_ptr (syntax_cmd_list, var_str ("lterm")))
      call parse_node_append_sub (pn_lsinglet, pn_lterm)
      pn_record => parse_node_get_sub_ptr (cmd%pn)
      call parse_node_append_sub (pn_lterm, pn_record)
      cmd%pn_lexpr => pn_lexpr
    end subroutine cmd_record_compile
```

Command execution. Again, transfer this to the embedded expression and just forget the logical result.

⟨*Commands: cmd record: TBP*⟩+≡
```
    procedure :: execute => cmd_record_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_record_execute (cmd, global)
      class(cmd_record_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      logical :: lval
      lval = eval_log (cmd%pn_lexpr, global%var_list)
    end subroutine cmd_record_execute
```

Unstable particles

Mark a particle as unstable. For each unstable particle, we store a number of decay channels and compute their respective BRs.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_unstable_t
      private
      integer :: n_proc = 0
      type(string_t), dimension(:), allocatable :: process_id
      type(parse_node_t), pointer :: pn_prt_in => null ()
     contains
     ⟨Commands: cmd unstable: TBP⟩
    end type cmd_unstable_t
```

Output: we know the process IDs.

⟨*Commands: cmd unstable: TBP*⟩≡
```
    procedure :: write => cmd_unstable_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_unstable_write (cmd, unit, indent)
      class(cmd_unstable_t), intent(in) :: cmd
```

```
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,1x,I0,1x,A)", advance="no")  &
           "unstable:", 1, "("
      do i = 1, cmd%n_proc
         if (i > 1)  write (u, "(A,1x)", advance="no")  ","
         write (u, "(A)", advance="no")  char (cmd%process_id(i))
      end do
      write (u, "(A)")  ")"
   end subroutine cmd_unstable_write
```

Compile. Initiate an eval tree for the decaying particle and determine the decay channel process IDs.

⟨*Commands: cmd unstable: TBP*⟩+≡
```
   procedure :: compile => cmd_unstable_compile
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_unstable_compile (cmd, global)
     class(cmd_unstable_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), pointer :: pn_list, pn_proc
     integer :: i
     cmd%pn_prt_in => parse_node_get_sub_ptr (cmd%pn, 2)
     pn_list => parse_node_get_next_ptr (cmd%pn_prt_in)
     if (associated (pn_list)) then
        select case (char (parse_node_get_rule_key (pn_list)))
        case ("unstable_arg")
           cmd%n_proc = parse_node_get_n_sub (pn_list)
           cmd%pn_opt => parse_node_get_next_ptr (pn_list)
        case default
           cmd%n_proc = 0
           cmd%pn_opt => pn_list
           pn_list => null ()
        end select
     end if
     call cmd%compile_options (global)
     if (associated (pn_list)) then
        allocate (cmd%process_id (cmd%n_proc))
        pn_proc => parse_node_get_sub_ptr (pn_list)
        do i = 1, cmd%n_proc
           cmd%process_id(i) = parse_node_get_string (pn_proc)
           call cmd%local%process_stack%init_result_vars (cmd%process_id(i))
           pn_proc => parse_node_get_next_ptr (pn_proc)
        end do
     else
        allocate (cmd%process_id (0))
     end if
   end subroutine cmd_unstable_compile
```

Command execution. Evaluate the decaying particle and mark the decays in the current model object.

3080

⟨*Commands: cmd unstable: TBP*⟩+≡
```
    procedure :: execute => cmd_unstable_execute
```
⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_unstable_execute (cmd, global)
      class(cmd_unstable_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      logical :: auto_decays, auto_decays_radiative
      integer :: auto_decays_multiplicity
      logical :: isotropic_decay, diagonal_decay
      type(pdg_array_t) :: pa_in
      integer :: pdg_in
      type(string_t) :: libname_cur, libname_dec
      type(string_t), dimension(:), allocatable :: auto_id, tmp_id
      integer :: n_proc_user
      integer :: i, u_tmp
      character(80) :: buffer
      associate (var_list => cmd%local%var_list)
        auto_decays = &
             var_list_get_lval (var_list, var_str ("?auto_decays"))
        if (auto_decays) then
           auto_decays_multiplicity = &
                var_list_get_ival (var_list, var_str ("auto_decays_multiplicity"))
           auto_decays_radiative = &
                var_list_get_lval (var_list, var_str ("?auto_decays_radiative"))
        end if
        isotropic_decay = &
             var_list_get_lval (var_list, var_str ("?isotropic_decay"))
        if (.not. isotropic_decay) then
           diagonal_decay = &
                var_list_get_lval (var_list, var_str ("?diagonal_decay"))
        else
           diagonal_decay = .false.
        end if
      end associate
      pa_in = eval_pdg_array (cmd%pn_prt_in, cmd%local%var_list)
      if (pdg_array_get_length (pa_in) /= 1) &
           call msg_fatal ("Unstable: decaying particle must be unique")
      pdg_in = pdg_array_get (pa_in, 1)
      n_proc_user = cmd%n_proc
      if (auto_decays) then
         call create_auto_decays (pdg_in, &
              auto_decays_multiplicity, auto_decays_radiative, &
              libname_dec, auto_id, cmd%local)
         allocate (tmp_id (cmd%n_proc + size (auto_id)))
         tmp_id(:cmd%n_proc) = cmd%process_id
         tmp_id(cmd%n_proc+1:) = auto_id
         call move_alloc (from = tmp_id, to = cmd%process_id)
         cmd%n_proc = size (cmd%process_id)
      end if
      libname_cur = cmd%local%prclib%get_name ()
      do i = 1, cmd%n_proc
         if (i == n_proc_user + 1) then
            call cmd%local%update_prclib &
                 (cmd%local%prclib_stack%get_library_ptr (libname_dec))
```

```
            end if
            if (.not. cmd%local%process_stack%exists (cmd%process_id(i))) then
               call var_list_set_log (cmd%local%var_list, &
                    var_str ("?decay_rest_frame"), .false., is_known = .true.)
               call integrate_process (cmd%process_id(i), cmd%local)
               call cmd%local%process_stack%fill_result_vars (cmd%process_id(i))
            end if
         end do
         call cmd%local%update_prclib &
              (cmd%local%prclib_stack%get_library_ptr (libname_cur))
         call cmd%local%modify_particle (pdg_in, stable = .false., &
              decay = cmd%process_id, &
              isotropic_decay = isotropic_decay, &
              diagonal_decay = diagonal_decay, &
              polarized = .false.)
         u_tmp = free_unit ()
         open (u_tmp, status = "scratch", action = "readwrite")
         call show_unstable (cmd%local, pdg_in, u_tmp)
         rewind (u_tmp)
         do
            read (u_tmp, "(A)", end = 1)  buffer
            write (msg_buffer, "(A)")  trim (buffer)
            call msg_message ()
         end do
1        continue
         close (u_tmp)
      end subroutine cmd_unstable_execute
```

Show data for the current unstable particle. This is called both by the `unstable` and by the `show` command.

⟨*Commands: procedures*⟩+≡

```
      subroutine show_unstable (global, pdg, u)
        type(rt_data_t), intent(in), target :: global
        integer, intent(in) :: pdg, u
        type(flavor_t) :: flv
        type(string_t), dimension(:), allocatable :: decay
        real(default), dimension(:), allocatable :: br
        real(default) :: width
        type(process_t), pointer :: process
        type(process_component_def_t), pointer :: prc_def
        type(string_t), dimension(:), allocatable :: prt_out, prt_out_str
        integer :: i, j
        call flavor_init (flv, pdg, global%model)
        call flavor_get_decays (flv, decay)
        if (.not. allocated (decay))  return
        allocate (prt_out_str (size (decay)))
        allocate (br (size (decay)))
        do i = 1, size (br)
           process => global%process_stack%get_process_ptr (decay(i))
           prc_def => process%get_component_def_ptr (1)
           call prc_def%get_prt_out (prt_out)
           prt_out_str(i) = prt_out(1)
           do j = 2, size (prt_out)
```

```
              prt_out_str(i) = prt_out_str(i) // ", " // prt_out(j)
           end do
           br(i) = process%get_integral ()
        end do
     if (all (br >= 0)) then
        if (any (br > 0)) then
           width = sum (br)
           br = br / sum (br)
           write (u, "(A)") "Unstable particle " &
                // char (flavor_get_name (flv)) &
                // ": computed branching ratios:"
           do i = 1, size (br)
              write (u, "(2x,A,':'," // FMT_14 // ",3x,A)") &
                   char (decay(i)), br(i), char (prt_out_str(i))
           end do
           write (u, "(2x,'Total width ='," // FMT_14 // ",' GeV (computed)')")  width
           write (u, "(2x,'            ='," // FMT_14 // ",' GeV (preset)')") &
                flavor_get_width (flv)
           if (flavor_decays_isotropically (flv)) then
              write (u, "(2x,A)")  "Decay options: isotropic"
           else if (flavor_decays_diagonal (flv)) then
              write (u, "(2x,A)")  "Decay options: &
                   &projection on diagonal helicity states"
           else
              write (u, "(2x,A)")  "Decay options: helicity treated exactly"
           end if
        else
           call msg_fatal ("Unstable particle " &
                // char (flavor_get_name (flv)) &
                // ": partial width vanishes for all decay channels")
        end if
     else
        call msg_fatal ("Unstable particle " &
                // char (flavor_get_name (flv)) &
                // ": partial width is negative")
     end if
  end subroutine show_unstable
```

Auto decays: create process IDs and make up process configurations, using the PDG codes generated by the `fs_table` make method.

We allocate and use a self-contained process library that contains only the decay processes of the current particle. When done, we revert the global library pointer to the original library but return the name of the new one. The new library becomes part of the global library stack and can thus be referred to at any time.

⟨*Commands: procedures*⟩+≡
```
  subroutine create_auto_decays &
       (pdg_in, mult, rad, libname_dec, process_id, global)
    integer, intent(in) :: pdg_in
    integer, intent(in) :: mult
    logical, intent(in) :: rad
    type(string_t), intent(out) :: libname_dec
    type(string_t), dimension(:), allocatable, intent(out) :: process_id
```

```
type(rt_data_t), intent(inout) :: global
type(prclib_entry_t), pointer :: lib_entry
type(process_library_t), pointer :: lib
type(fs_table_t) :: fs_table
integer, dimension(:), allocatable :: pdg_out
character(80) :: buffer
character :: p_or_a
type(string_t) :: process_string, libname_cur
type(flavor_t) :: flv_in, flv_out
type(string_t) :: prt_in
type(string_t), dimension(:), allocatable :: prt_out
type(process_configuration_t) :: prc_config
integer :: i, j
call fs_table%make (global%model, pdg_in, mult, rad)
call flavor_init (flv_in, pdg_in, global%model)
prt_in = flavor_get_name (flv_in)
if (pdg_in > 0) then
   p_or_a = "p"
else
   p_or_a = "a"
end if
call msg_message ("Creating decay process library for particle " &
     // char (prt_in))
libname_cur = global%prclib%get_name ()
write (buffer, "(A,A,I0)")  "_d", p_or_a, abs (pdg_in)
libname_dec = libname_cur // trim (buffer)
lib => global%prclib_stack%get_library_ptr (libname_dec)
if (.not. (associated (lib))) then
   allocate (lib_entry)
   call lib_entry%init (libname_dec)
   lib => lib_entry%process_library_t
   call global%add_prclib (lib_entry)
else
   call global%update_prclib (lib)
end if
allocate (process_id (fs_table%get_length ()))
do i = 1, size (process_id)
   write (buffer, "(A,'_',A,I0,'_',I0)")  "decay", p_or_a, abs (pdg_in), i
   process_id(i) = trim (buffer)
   process_string = process_id(i) // ": " // prt_in // " =>"
   call fs_table%get_pdg_out (i, pdg_out)
   allocate (prt_out (size (pdg_out)))
   do j = 1, size (pdg_out)
      call flavor_init (flv_out, pdg_out(j), global%model)
      prt_out(j) = flavor_get_name (flv_out)
      process_string = process_string // " " // prt_out(j)
   end do
   call msg_message (char (process_string))
   call prc_config%init (process_id(i), 1, 1, global)
   call prc_config%setup_component (1, &
        new_prt_spec ([prt_in]), new_prt_spec (prt_out), global)
   call prc_config%record (global)
   deallocate (prt_out)
   deallocate (pdg_out)
```

```
          end do
          lib => global%prclib_stack%get_library_ptr (libname_cur)
          call global%update_prclib (lib)
        end subroutine create_auto_decays
```

(Stable particles

Revert the unstable declaration for a list of particles.

⟨Commands: types⟩+≡
```
    type, extends (command_t) :: cmd_stable_t
      private
      type(parse_node_p), dimension(:), allocatable :: pn_pdg
     contains
     ⟨Commands: cmd stable: TBP⟩
    end type cmd_stable_t
```

Output: we know only the number of particles.

⟨Commands: cmd stable: TBP⟩≡
```
    procedure :: write => cmd_stable_write
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_stable_write (cmd, unit, indent)
      class(cmd_stable_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,1x,I0)")  "stable:", size (cmd%pn_pdg)
    end subroutine cmd_stable_write
```

Compile. Assign parse nodes for the particle IDs.

⟨Commands: cmd stable: TBP⟩+≡
```
    procedure :: compile => cmd_stable_compile
```

⟨Commands: procedures⟩+≡
```
    subroutine cmd_stable_compile (cmd, global)
      class(cmd_stable_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_list, pn_prt
      integer :: n, i
      pn_list => parse_node_get_sub_ptr (cmd%pn, 2)
      cmd%pn_opt => parse_node_get_next_ptr (pn_list)
      call cmd%compile_options (global)
      n = parse_node_get_n_sub (pn_list)
      allocate (cmd%pn_pdg (n))
      pn_prt => parse_node_get_sub_ptr (pn_list)
      i = 1
      do while (associated (pn_prt))
         cmd%pn_pdg(i)%ptr => pn_prt
         pn_prt  => parse_node_get_next_ptr (pn_prt)
         i = i + 1
      end do
```

```
          end subroutine cmd_stable_compile
```

Execute: apply the modifications to the current model.

⟨*Commands: cmd stable: TBP*⟩+≡
```
    procedure :: execute => cmd_stable_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_stable_execute (cmd, global)
      class(cmd_stable_t), intent(inout) :: cmd
      type(rt_data_t), target, intent(inout) :: global
      type(pdg_array_t) :: pa
      integer :: pdg
      type(flavor_t) :: flv
      integer :: i
      do i = 1, size (cmd%pn_pdg)
         pa = eval_pdg_array (cmd%pn_pdg(i)%ptr, cmd%local%var_list)
         if (pdg_array_get_length (pa) /= 1) &
              call msg_fatal ("Stable: listed particles must be unique")
         pdg = pdg_array_get (pa, 1)
         call cmd%local%modify_particle (pdg, stable = .true., &
           isotropic_decay = .false., &
           diagonal_decay = .false., &
           polarized = .false.)
         call flavor_init (flv, pdg, cmd%local%model)
         call msg_message ("Particle " &
              // char (flavor_get_name (flv)) &
              // " declared as stable")
      end do
    end subroutine cmd_stable_execute
```

Polarized particles

These commands mark particles as (un)polarized, to be applied in subsequent simulation passes. Since this is technically the same as the `stable` command, we take a shortcut and make this an extension, just overriding methods.

⟨*Commands: types*⟩+≡
```
    type, extends (cmd_stable_t) :: cmd_polarized_t
     contains
     ⟨Commands: cmd polarized: TBP⟩
    end type cmd_polarized_t

    type, extends (cmd_stable_t) :: cmd_unpolarized_t
     contains
     ⟨Commands: cmd unpolarized: TBP⟩
    end type cmd_unpolarized_t
```

Output: we know only the number of particles.

⟨*Commands: cmd polarized: TBP*⟩≡
```
    procedure :: write => cmd_polarized_write
```

⟨*Commands: cmd unpolarized: TBP*⟩≡
```
    procedure :: write => cmd_unpolarized_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_polarized_write (cmd, unit, indent)
  class(cmd_polarized_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,I0)")  "polarized:", size (cmd%pn_pdg)
end subroutine cmd_polarized_write

subroutine cmd_unpolarized_write (cmd, unit, indent)
  class(cmd_unpolarized_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,I0)")  "unpolarized:", size (cmd%pn_pdg)
end subroutine cmd_unpolarized_write
```

Compile: accounted for by the base command.
    Execute: apply the modifications to the current model.

⟨*Commands: cmd polarized: TBP*⟩+≡

```
procedure :: execute => cmd_polarized_execute
```

⟨*Commands: cmd unpolarized: TBP*⟩+≡

```
procedure :: execute => cmd_unpolarized_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_polarized_execute (cmd, global)
  class(cmd_polarized_t), intent(inout) :: cmd
  type(rt_data_t), target, intent(inout) :: global
  type(pdg_array_t) :: pa
  integer :: pdg
  type(flavor_t) :: flv
  integer :: i
  do i = 1, size (cmd%pn_pdg)
     pa = eval_pdg_array (cmd%pn_pdg(i)%ptr, cmd%local%var_list)
     if (pdg_array_get_length (pa) /= 1) &
          call msg_fatal ("Polarized: listed particles must be unique")
     pdg = pdg_array_get (pa, 1)
     call cmd%local%modify_particle (pdg, polarized = .true., &
          stable = .true., &
          isotropic_decay = .false., &
          diagonal_decay = .false.)
     call flavor_init (flv, pdg, cmd%local%model)
     call msg_message ("Particle " &
          // char (flavor_get_name (flv)) &
          // " declared as polarized")
  end do
end subroutine cmd_polarized_execute

subroutine cmd_unpolarized_execute (cmd, global)
  class(cmd_unpolarized_t), intent(inout) :: cmd
  type(rt_data_t), target, intent(inout) :: global
  type(pdg_array_t) :: pa
```

```
      integer :: pdg
      type(flavor_t) :: flv
      integer :: i
      do i = 1, size (cmd%pn_pdg)
         pa = eval_pdg_array (cmd%pn_pdg(i)%ptr, cmd%local%var_list)
         if (pdg_array_get_length (pa) /= 1) &
              call msg_fatal ("Unpolarized: listed particles must be unique")
         pdg = pdg_array_get (pa, 1)
         call cmd%local%modify_particle (pdg, polarized = .false., &
              stable = .true., &
              isotropic_decay = .false., &
              diagonal_decay = .false.)
         call flavor_init (flv, pdg, cmd%local%model)
         call msg_message ("Particle " &
              // char (flavor_get_name (flv)) &
              // " declared as unpolarized")
      end do
    end subroutine cmd_unpolarized_execute
```

Parameters: formats for event-sample output

Specify all event formats that are to be used for output files in the subsequent simulation run. (The raw format is on by default and can be turned off here.)

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_sample_format_t
      private
      type(string_t), dimension(:), allocatable :: format
    contains
    ⟨Commands: cmd sample format: TBP⟩
    end type cmd_sample_format_t
```

Output: here, everything is known.

⟨*Commands: cmd sample format: TBP*⟩≡
```
    procedure :: write => cmd_sample_format_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_sample_format_write (cmd, unit, indent)
      class(cmd_sample_format_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "sample_format = "
      do i = 1, size (cmd%format)
         if (i > 1)  write (u, "(A,1x)", advance="no")  ","
         write (u, "(A)", advance="no")  char (cmd%format(i))
      end do
      write (u, "(A)")
    end subroutine cmd_sample_format_write
```

Compile. Initialize evaluation trees.

⟨*Commands: cmd sample format: TBP*⟩+≡
```
    procedure :: compile => cmd_sample_format_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_sample_format_compile (cmd, global)
      class(cmd_sample_format_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg
      type(parse_node_t), pointer :: pn_format
      integer :: i, n_format
      pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
      if (associated (pn_arg)) then
         n_format = parse_node_get_n_sub (pn_arg)
         allocate (cmd%format (n_format))
         pn_format => parse_node_get_sub_ptr (pn_arg)
         i = 0
         do while (associated (pn_format))
            i = i + 1
            cmd%format(i) = parse_node_get_string (pn_format)
            pn_format => parse_node_get_next_ptr (pn_format)
         end do
      else
         allocate (cmd%format (0))
      end if
    end subroutine cmd_sample_format_compile
```

Execute. Transfer the list of format specifications to the corresponding array in the runtime data set.

⟨*Commands: cmd sample format: TBP*⟩+≡
```
    procedure :: execute => cmd_sample_format_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_sample_format_execute (cmd, global)
      class(cmd_sample_format_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      if (allocated (global%sample_fmt))  deallocate (global%sample_fmt)
      allocate (global%sample_fmt (size (cmd%format)), source = cmd%format)
    end subroutine cmd_sample_format_execute
```

The simulate command

This is the actual SINDARIN command.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_simulate_t
      ! not private anymore as required by the whizard-c-interface
      integer :: n_proc = 0
      type(string_t), dimension(:), allocatable :: process_id
     contains
     ⟨Commands: cmd simulate: TBP⟩
    end type cmd_simulate_t
```

Output: we know the process IDs.

⟨*Commands: cmd simulate: TBP*⟩≡
```
    procedure :: write => cmd_simulate_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_simulate_write (cmd, unit, indent)
      class(cmd_simulate_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u, i
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A)", advance="no")  "simulate ("
      do i = 1, cmd%n_proc
         if (i > 1)  write (u, "(A,1x)", advance="no")  ","
         write (u, "(A)", advance="no")  char (cmd%process_id(i))
      end do
      write (u, "(A)")  ")"
    end subroutine cmd_simulate_write
```

Compile. In contrast to WHIZARD 1 the confusing option to give the number
of unweighted events for weighted events as if unweighting were to take place has
been abandoned. (We both use `n_events` for weighted and unweighted events,
the variable `n_calls` from WHIZARD 1 has been discarded.

⟨*Commands: cmd simulate: TBP*⟩+≡
```
    procedure :: compile => cmd_simulate_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_simulate_compile (cmd, global)
      class(cmd_simulate_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_proclist, pn_proc
      integer :: i
      pn_proclist => parse_node_get_sub_ptr (cmd%pn, 2)
      cmd%pn_opt => parse_node_get_next_ptr (pn_proclist)
      call cmd%compile_options (global)
      cmd%n_proc = parse_node_get_n_sub (pn_proclist)
      allocate (cmd%process_id (cmd%n_proc))
      pn_proc => parse_node_get_sub_ptr (pn_proclist)
      do i = 1, cmd%n_proc
         cmd%process_id(i) = parse_node_get_string (pn_proc)
         call global%process_stack%init_result_vars (cmd%process_id(i))
         pn_proc => parse_node_get_next_ptr (pn_proc)
      end do
    end subroutine cmd_simulate_compile
```

Execute command: Simulate events. This is done via a `simulation_t` object
and its associated methods.

Signal handling: the `generate` method may exit abnormally if there is a
pending signal. The current logic ensures that the `es_array` output channels
are closed before the `execute` routine returns. The program will terminate then
in `command_list_execute`.

⟨*Commands: cmd simulate: TBP*⟩+≡
```
    procedure :: execute => cmd_simulate_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_simulate_execute (cmd, global)
  class(cmd_simulate_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(rt_data_t), dimension(:), allocatable, target :: alt_env
  integer :: n_events, n_fmt
  type(string_t) :: sample
  logical :: rebuild_events, read_raw, write_raw
  type(simulation_t), target :: sim
  type(string_t), dimension(:), allocatable :: sample_fmt
  type(event_stream_array_t) :: es_array
  type(event_sample_data_t) :: data
  integer :: i, checkpoint
  if (allocated (cmd%local%pn%alt_setup)) then
     allocate (alt_env (size (cmd%local%pn%alt_setup)))
     do i = 1, size (alt_env)
        call build_alt_setup (alt_env(i), cmd%local, &
             cmd%local%pn%alt_setup(i)%ptr)
     end do
     call sim%init (cmd%process_id, .true., .true., cmd%local, alt_env)
  else
     call sim%init (cmd%process_id, .true., .true., cmd%local)
  end if
  if (signal_is_pending ())  return
  if (sim%is_valid ()) then
     call sim%init_process_selector ()
     call openmp_set_num_threads_verbose &
          (var_list_get_ival (cmd%local%var_list, "openmp_num_threads"), &
          var_list_get_lval (cmd%local%var_list, "?openmp_logging"))
     call sim%compute_n_events (n_events, cmd%local%var_list)
     sample = var_list_get_sval (cmd%local%var_list, var_str ("$sample"))
     if (sample == "")  sample = sim%get_default_sample_name ()
     rebuild_events = &
          var_list_get_lval (cmd%local%var_list, var_str ("?rebuild_events"))
     read_raw = &
          var_list_get_lval (cmd%local%var_list, var_str ("?read_raw")) &
          .and. .not. rebuild_events
     write_raw = &
          var_list_get_lval (cmd%local%var_list, var_str ("?write_raw"))
     checkpoint = &
          var_list_get_ival (cmd%local%var_list, var_str ("checkpoint"))
     if (read_raw) then
        inquire (file = char (sample) // ".evx", exist = read_raw)
     end if
     if (allocated (cmd%local%sample_fmt)) then
        n_fmt = size (cmd%local%sample_fmt)
     else
        n_fmt = 0
     end if
     data = sim%get_data ()
     data%n_evt = n_events
     if (read_raw) then
        allocate (sample_fmt (n_fmt))
        if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
```

3091

```
            call es_array%init (sample, &
                 sample_fmt, sim%get_process_ptr (), cmd%local, &
                 data = data, &
                 input = var_str ("raw"), &
                 allow_switch = write_raw, &
                 checkpoint = checkpoint)
            call sim%generate (n_events, es_array)
            call es_array%final ()
         else if (write_raw) then
            allocate (sample_fmt (n_fmt + 1))
            if (n_fmt > 0)  sample_fmt(:n_fmt) = cmd%local%sample_fmt
            sample_fmt(n_fmt+1) = var_str ("raw")
            call es_array%init (sample, &
                 sample_fmt, sim%get_process_ptr (), cmd%local, &
                 data = data, &
                 checkpoint = checkpoint)
            call sim%generate (n_events, es_array)
            call es_array%final ()
         else if (allocated (cmd%local%sample_fmt) .or. checkpoint > 0) then
            allocate (sample_fmt (n_fmt))
            if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
            call es_array%init (sample, &
                 sample_fmt, sim%get_process_ptr (), cmd%local, &
                 data = data, &
                 checkpoint = checkpoint)
            call sim%generate (n_events, es_array)
            call es_array%final ()
         else
            call sim%generate (n_events)
         end if
      end if
      call sim%final ()
   end subroutine cmd_simulate_execute
```

Build an alternative setup: the parse tree is stored in the global environment.
We create a temporary command list to compile and execute this; the result
is an alternative local environment `alt_env` which we can hand over to the
`simulate` command.

⟨*Commands: procedures*⟩+≡
```
   recursive subroutine build_alt_setup (alt_env, global, pn)
     type(rt_data_t), intent(inout), target :: alt_env
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), intent(in), target :: pn
     type(command_list_t), allocatable :: alt_options
     allocate (alt_options)
     call alt_env%local_init (global)
     call alt_env%link (global)
     call alt_options%compile (pn, alt_env)
     call alt_options%execute (alt_env)
     call global%restore (alt_env, keep_local = .true.)
     call alt_options%final ()
   end subroutine build_alt_setup
```

The rescan command

This is the actual SINDARIN command.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_rescan_t
  ! private
  type(parse_node_t), pointer :: pn_filename => null ()
  integer :: n_proc = 0
  type(string_t), dimension(:), allocatable :: process_id
 contains
 ⟨Commands: cmd rescan: TBP⟩
end type cmd_rescan_t
```

Output: we know the process IDs.

⟨*Commands: cmd rescan: TBP*⟩≡
```
procedure :: write => cmd_rescan_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_rescan_write (cmd, unit, indent)
  class(cmd_rescan_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no")  "rescan ("
  do i = 1, cmd%n_proc
     if (i > 1)  write (u, "(A,1x)", advance="no")  ","
     write (u, "(A)", advance="no")  char (cmd%process_id(i))
  end do
  write (u, "(A)")  ")"
end subroutine cmd_rescan_write
```

Compile. The command takes a suffix argument, namely the file name of requested event file.

⟨*Commands: cmd rescan: TBP*⟩+≡
```
procedure :: compile => cmd_rescan_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_rescan_compile (cmd, global)
  class(cmd_rescan_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_filename, pn_proclist, pn_proc
  integer :: i
  pn_filename => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_proclist => parse_node_get_next_ptr (pn_filename)
  cmd%pn_opt => parse_node_get_next_ptr (pn_proclist)
  call cmd%compile_options (global)
  cmd%pn_filename => pn_filename
  cmd%n_proc = parse_node_get_n_sub (pn_proclist)
  allocate (cmd%process_id (cmd%n_proc))
  pn_proc => parse_node_get_sub_ptr (pn_proclist)
  do i = 1, cmd%n_proc
     cmd%process_id(i) = parse_node_get_string (pn_proc)
     pn_proc => parse_node_get_next_ptr (pn_proc)
```

```
      end do
    end subroutine cmd_rescan_compile
```

Execute command: Rescan events. This is done via a `simulation_t` object and its associated methods.

⟨*Commands: cmd rescan: TBP*⟩+≡
```
    procedure :: execute => cmd_rescan_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_rescan_execute (cmd, global)
      class(cmd_rescan_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(rt_data_t), dimension(:), allocatable, target :: alt_env
      type(string_t) :: sample
      logical :: exist, write_raw, update_event, update_sqme, update_weight
      logical :: recover_beams
      type(simulation_t), target :: sim
      type(event_sample_data_t) :: input_data, data
      type(string_t) :: input_sample
      integer :: n_fmt
      type(string_t), dimension(:), allocatable :: sample_fmt
      type(string_t) :: input_format, input_ext, input_file
      type(string_t) :: lhef_extension
      type(event_stream_array_t) :: es_array
      integer :: i, n_events
      if (allocated (cmd%local%pn%alt_setup)) then
         allocate (alt_env (size (cmd%local%pn%alt_setup)))
         do i = 1, size (alt_env)
            call build_alt_setup (alt_env(i), cmd%local, &
                 cmd%local%pn%alt_setup(i)%ptr)
         end do
         call sim%init (cmd%process_id, .false., .false., cmd%local, alt_env)
      else
         call sim%init (cmd%process_id, .false., .false., cmd%local)
      end if
      call sim%compute_n_events (n_events, cmd%local%var_list)
      input_sample = eval_string (cmd%pn_filename, cmd%local%var_list)
      input_format = var_list_get_sval (cmd%local%var_list, &
           var_str ("$rescan_input_format"))
      sample = var_list_get_sval (cmd%local%var_list, var_str ("$sample"))
      if (sample == "")  sample = sim%get_default_sample_name ()
      write_raw = var_list_get_lval (cmd%local%var_list, var_str ("?write_raw"))
      if (allocated (cmd%local%sample_fmt)) then
         n_fmt = size (cmd%local%sample_fmt)
      else
         n_fmt = 0
      end if
      if (write_raw) then
         if (sample == input_sample) then
            call msg_error ("Rescan: ?write_raw = true: " &
                 // "suppressing raw event output (filename clashes with input)")
            allocate (sample_fmt (n_fmt))
            if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
         else
```

```
            allocate (sample_fmt (n_fmt + 1))
            if (n_fmt > 0)  sample_fmt(:n_fmt) = cmd%local%sample_fmt
            sample_fmt(n_fmt+1) = var_str ("raw")
       end if
   else
      allocate (sample_fmt (n_fmt))
      if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
   end if
   update_event = &
        var_list_get_lval (cmd%local%var_list, var_str ("?update_event"))
   update_sqme = &
        var_list_get_lval (cmd%local%var_list, var_str ("?update_sqme"))
   update_weight = &
        var_list_get_lval (cmd%local%var_list, var_str ("?update_weight"))
   recover_beams = &
        var_list_get_lval (cmd%local%var_list, var_str ("?recover_beams"))
   if (update_event .or. update_sqme) then
      call msg_message ("Recalculating observables")
      if (update_sqme) then
         call msg_message ("Recalculating squared matrix elements")
      end if
   end if
   lhef_extension = &
        var_list_get_sval (cmd%local%var_list, var_str ("$lhef_extension"))
   select case (char (input_format))
   case ("raw");  input_ext = "evx"
   case ("lhef"); input_ext = lhef_extension
   case default
      call msg_fatal ("rescan: input sample format '" // char (input_format) &
           // "' not supported")
   end select
   input_file = input_sample // "." // input_ext
   inquire (file = char (input_file), exist = exist)
   if (exist) then
      input_data = sim%get_data (alt = .false.)
      input_data%n_evt = n_events
      data = sim%get_data ()
      data%n_evt = n_events
      input_data%md5sum_cfg = ""
      call es_array%init (sample, &
           sample_fmt, sim%get_process_ptr (), cmd%local, data, &
           input = input_format, input_sample = input_sample, &
           input_data = input_data, &
           allow_switch = .false.)
      call sim%rescan (n_events, es_array, &
           update_event = update_event, &
           update_sqme = update_sqme, &
           update_weight = update_weight, &
           recover_beams = recover_beams, &
           global = cmd%local)
      call es_array%final ()
   else
      call msg_fatal ("Rescan: event file '" &
           // char (input_file) // "' not found")
```

```
      end if
      call sim%final ()
    end subroutine cmd_rescan_execute
```

Parameters: number of iterations

Specify number of iterations and number of calls for one integration pass.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_iterations_t
      private
      integer :: n_pass = 0
      type(parse_node_p), dimension(:), allocatable :: pn_expr_n_it
      type(parse_node_p), dimension(:), allocatable :: pn_expr_n_calls
      type(parse_node_p), dimension(:), allocatable :: pn_sexpr_adapt
    contains
    ⟨Commands: cmd iterations: TBP⟩
    end type cmd_iterations_t
```

Output. Display the number of passes, which is known after compilation.

⟨*Commands: cmd iterations: TBP*⟩≡
```
    procedure :: write => cmd_iterations_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_iterations_write (cmd, unit, indent)
      class(cmd_iterations_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      select case (cmd%n_pass)
      case (0)
        write (u, "(1x,A)")  "iterations: [empty]"
      case (1)
        write (u, "(1x,A,I0,A)")  "iterations: ", cmd%n_pass, " pass"
      case default
        write (u, "(1x,A,I0,A)")  "iterations: ", cmd%n_pass, " passes"
      end select
    end subroutine cmd_iterations_write
```

Compile. Initialize evaluation trees.

⟨*Commands: cmd iterations: TBP*⟩+≡
```
    procedure :: compile => cmd_iterations_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_iterations_compile (cmd, global)
      class(cmd_iterations_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg, pn_n_it, pn_n_calls, pn_adapt
      type(parse_node_t), pointer :: pn_it_spec, pn_calls_spec, pn_adapt_spec
      integer :: i
      pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
      if (associated (pn_arg)) then
```

```
      cmd%n_pass = parse_node_get_n_sub (pn_arg)
      allocate (cmd%pn_expr_n_it (cmd%n_pass))
      allocate (cmd%pn_expr_n_calls (cmd%n_pass))
      allocate (cmd%pn_sexpr_adapt (cmd%n_pass))
      pn_it_spec => parse_node_get_sub_ptr (pn_arg)
      i = 1
      do while (associated (pn_it_spec))
         pn_n_it => parse_node_get_sub_ptr (pn_it_spec)
         pn_calls_spec => parse_node_get_next_ptr (pn_n_it)
         pn_n_calls => parse_node_get_sub_ptr (pn_calls_spec, 2)
         pn_adapt_spec => parse_node_get_next_ptr (pn_calls_spec)
         if (associated (pn_adapt_spec)) then
            pn_adapt => parse_node_get_sub_ptr (pn_adapt_spec, 2)
         else
            pn_adapt => null ()
         end if
         cmd%pn_expr_n_it(i)%ptr => pn_n_it
         cmd%pn_expr_n_calls(i)%ptr => pn_n_calls
         cmd%pn_sexpr_adapt(i)%ptr => pn_adapt
         i = i + 1
         pn_it_spec => parse_node_get_next_ptr (pn_it_spec)
      end do
   else
      allocate (cmd%pn_expr_n_it (0))
      allocate (cmd%pn_expr_n_calls (0))
   end if
 end subroutine cmd_iterations_compile
```

Execute. Evaluate the trees and transfer the results to the iteration list in the
runtime data set.

⟨*Commands: cmd iterations: TBP*⟩+≡
```
    procedure :: execute => cmd_iterations_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_iterations_execute (cmd, global)
      class(cmd_iterations_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      integer, dimension(cmd%n_pass) :: n_it, n_calls
      logical, dimension(cmd%n_pass) :: custom_adapt
      type(string_t), dimension(cmd%n_pass) :: adapt_code
      integer :: i
      do i = 1, cmd%n_pass
         n_it(i) = eval_int (cmd%pn_expr_n_it(i)%ptr, global%var_list)
         n_calls(i) = &
              eval_int (cmd%pn_expr_n_calls(i)%ptr, global%var_list)
         if (associated (cmd%pn_sexpr_adapt(i)%ptr)) then
            adapt_code(i) = &
                 eval_string (cmd%pn_sexpr_adapt(i)%ptr, &
                              global%var_list, is_known = custom_adapt(i))
         else
            custom_adapt(i) = .false.
         end if
      end do
      call global%it_list%init (n_it, n_calls, custom_adapt, adapt_code)
```

```
        end subroutine cmd_iterations_execute
```

Range expressions

We need a special type for storing and evaluating range expressions.

⟨*Commands: parameters*⟩≡
```
    integer, parameter :: STEP_NONE = 0
    integer, parameter :: STEP_ADD = 1
    integer, parameter :: STEP_SUB = 2
    integer, parameter :: STEP_MUL = 3
    integer, parameter :: STEP_DIV = 4
    integer, parameter :: STEP_COMP_ADD = 11
    integer, parameter :: STEP_COMP_MUL = 13
```

There is an abstract base type and two implementations: scan over integers and scan over reals.

⟨*Commands: types*⟩+≡
```
    type, abstract :: range_t
      type(parse_node_t), pointer :: pn_expr => null ()
      type(parse_node_t), pointer :: pn_term => null ()
      type(parse_node_t), pointer :: pn_factor => null ()
      type(parse_node_t), pointer :: pn_value => null ()
      type(parse_node_t), pointer :: pn_literal => null ()
      type(parse_node_t), pointer :: pn_beg => null ()
      type(parse_node_t), pointer :: pn_end => null ()
      type(parse_node_t), pointer :: pn_step => null ()
      type(eval_tree_t) :: expr_beg
      type(eval_tree_t) :: expr_end
      type(eval_tree_t) :: expr_step
      integer :: step_mode = 0
      integer :: n_step = 0
    contains
    ⟨Commands: range: TBP⟩
    end type range_t
```

These are the implementations:

⟨*Commands: types*⟩+≡
```
    type, extends (range_t) :: range_int_t
      integer :: i_beg = 0
      integer :: i_end = 0
      integer :: i_step = 0
    contains
    ⟨Commands: range int: TBP⟩
  end type range_int_t

    type, extends (range_t) :: range_real_t
      real(default) :: r_beg = 0
      real(default) :: r_end = 0
      real(default) :: r_step = 0
      real(default) :: lr_beg  = 0
      real(default) :: lr_end  = 0
      real(default) :: lr_step = 0
```

```
      contains
```
⟨*Commands: range real: TBP*⟩
```
  end type range_real_t
```

Finalize the allocated dummy node. The other nodes are just pointers.

⟨*Commands: range: TBP*⟩≡
```
    procedure :: final => range_final
```

⟨*Commands: procedures*⟩+≡
```
    subroutine range_final (object)
      class(range_t), intent(inout) :: object
      if (associated (object%pn_expr)) then
         call parse_node_final (object%pn_expr, recursive = .false.)
         call parse_node_final (object%pn_term, recursive = .false.)
         call parse_node_final (object%pn_factor, recursive = .false.)
         call parse_node_final (object%pn_value, recursive = .false.)
         call parse_node_final (object%pn_literal, recursive = .false.)
         deallocate (object%pn_expr)
         deallocate (object%pn_term)
         deallocate (object%pn_factor)
         deallocate (object%pn_value)
         deallocate (object%pn_literal)
      end if
    end subroutine range_final
```

Output.

⟨*Commands: range: TBP*⟩+≡
```
    procedure (range_write), deferred :: write
    procedure :: base_write => range_write
```

⟨*Commands: range int: TBP*⟩≡
```
    procedure :: write => range_int_write
```

⟨*Commands: range real: TBP*⟩≡
```
    procedure :: write => range_real_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine range_write (object, unit)
      class(range_t), intent(in) :: object
      integer, intent(in), optional :: unit
      integer :: u
      u = output_unit (unit)
      write (u, "(1x,A)")  "Range specification:"
      if (associated (object%pn_expr)) then
         write (u, "(1x,A)")  "Dummy value:"
         call parse_node_write_rec (object%pn_expr, u)
      end if
      if (associated (object%pn_beg)) then
         write (u, "(1x,A)")  "Initial value:"
         call parse_node_write_rec (object%pn_beg, u)
         call eval_tree_write (object%expr_beg, u)
         if (associated (object%pn_end)) then
            write (u, "(1x,A)")  "Final value:"
            call parse_node_write_rec (object%pn_end, u)
            call eval_tree_write (object%expr_end, u)
```

```
            if (associated (object%pn_step)) then
                write (u, "(1x,A)")  "Step value:"
                call parse_node_write_rec (object%pn_step, u)
                select case (object%step_mode)
                case (STEP_ADD);   write (u, "(1x,A)")  "Step mode: +"
                case (STEP_SUB);   write (u, "(1x,A)")  "Step mode: -"
                case (STEP_MUL);   write (u, "(1x,A)")  "Step mode: *"
                case (STEP_DIV);   write (u, "(1x,A)")  "Step mode: /"
                case (STEP_COMP_ADD); write (u, "(1x,A)")  "Division mode: +"
                case (STEP_COMP_MUL); write (u, "(1x,A)")  "Division mode: *"
                end select
            end if
        end if
    else
        write (u, "(1x,A)")  "Expressions: [undefined]"
    end if
  end subroutine range_write

  subroutine range_int_write (object, unit)
    class(range_int_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    call object%base_write (unit)
    write (u, "(1x,A)")  "Range parameters:"
    write (u, "(3x,A,I0)")  "i_beg  = ", object%i_beg
    write (u, "(3x,A,I0)")  "i_end  = ", object%i_end
    write (u, "(3x,A,I0)")  "i_step = ", object%i_step
    write (u, "(3x,A,I0)")  "n_step = ", object%n_step
  end subroutine range_int_write

  subroutine range_real_write (object, unit)
    class(range_real_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = output_unit (unit)
    call object%base_write (unit)
    write (u, "(1x,A)")  "Range parameters:"
    write (u, "(3x,A," // FMT_19 // ")")  "r_beg  = ", object%r_beg
    write (u, "(3x,A," // FMT_19 // ")")  "r_end  = ", object%r_end
    write (u, "(3x,A," // FMT_19 // ")")  "r_step = ", object%r_end
    write (u, "(3x,A,I0)")  "n_step = ", object%n_step
  end subroutine range_real_write
```

Initialize, given a range expression parse node. This is common to the implementations.

⟨*Commands: range: TBP*⟩+≡

```
    procedure :: init => range_init
```

⟨*Commands: procedures*⟩+≡

```
    subroutine range_init (range, pn)
      class(range_t), intent(out) :: range
      type(parse_node_t), intent(in), target :: pn
      type(parse_node_t), pointer :: pn_spec, pn_end, pn_step_spec, pn_op
```

3100

```
      select case (char (parse_node_get_rule_key (pn)))
      case ("expr")
      case ("range_expr")
         range%pn_beg => parse_node_get_sub_ptr (pn)
         pn_spec => parse_node_get_next_ptr (range%pn_beg)
         if (associated (pn_spec)) then
            pn_end => parse_node_get_sub_ptr (pn_spec, 2)
            range%pn_end => pn_end
            pn_step_spec => parse_node_get_next_ptr (pn_end)
            if (associated (pn_step_spec)) then
               pn_op => parse_node_get_sub_ptr (pn_step_spec)
               range%pn_step => parse_node_get_next_ptr (pn_op)
               select case (char (parse_node_get_rule_key (pn_op)))
               case ("/+");  range%step_mode = STEP_ADD
               case ("/-");  range%step_mode = STEP_SUB
               case ("/*");  range%step_mode = STEP_MUL
               case ("//");  range%step_mode = STEP_DIV
               case ("/+/");  range%step_mode = STEP_COMP_ADD
               case ("/*/");  range%step_mode = STEP_COMP_MUL
               case default
                  call range%write ()
                  call msg_bug ("Range: step mode not implemented")
               end select
            else
               range%step_mode = STEP_ADD
            end if
         else
            range%step_mode = STEP_NONE
         end if
         call range%create_value_node ()
      case default
         call msg_bug ("range expression: node type '" &
               // char (parse_node_get_rule_key (pn)) &
               // "' not implemented")
      end select
    end subroutine range_init
```

This method manually creates a parse node (actually, a cascade of parse nodes) that hold a constant value as a literal. The idea is that this node is inserted as the right-hand side of a fake variable assignment, which is prepended to each scan iteration. Before the variable assignment is compiled and executed, we can manually reset the value of the literal and thus pretend that the loop variable is assigned this value.

⟨*Commands: range: TBP*⟩+≡
```
    procedure :: create_value_node => range_create_value_node
```

⟨*Commands: procedures*⟩+≡
```
    subroutine range_create_value_node (range)
      class(range_t), intent(inout) :: range
      allocate (range%pn_literal)
      allocate (range%pn_value)
      select type (range)
      type is (range_int_t)
         call parse_node_create_value (range%pn_literal, &
```

3101

```
                syntax_get_rule_ptr (syntax_cmd_list, var_str ("integer_literal")),&
                ival = 0)
         call parse_node_create_branch (range%pn_value, &
                syntax_get_rule_ptr (syntax_cmd_list, var_str ("integer_value")))
      type is (range_real_t)
         call parse_node_create_value (range%pn_literal, &
                syntax_get_rule_ptr (syntax_cmd_list, var_str ("real_literal")),&
                rval = 0._default)
         call parse_node_create_branch (range%pn_value, &
                syntax_get_rule_ptr (syntax_cmd_list, var_str ("real_value")))
      class default
         call msg_bug ("range: create value node: type not implemented")
      end select
      call parse_node_append_sub (range%pn_value, range%pn_literal)
      call parse_node_freeze_branch (range%pn_value)
      allocate (range%pn_factor)
      call parse_node_create_branch (range%pn_factor, &
             syntax_get_rule_ptr (syntax_cmd_list, var_str ("factor")))
      call parse_node_append_sub (range%pn_factor, range%pn_value)
      call parse_node_freeze_branch (range%pn_factor)
      allocate (range%pn_term)
      call parse_node_create_branch (range%pn_term, &
             syntax_get_rule_ptr (syntax_cmd_list, var_str ("term")))
      call parse_node_append_sub (range%pn_term, range%pn_factor)
      call parse_node_freeze_branch (range%pn_term)
      allocate (range%pn_expr)
      call parse_node_create_branch (range%pn_expr, &
             syntax_get_rule_ptr (syntax_cmd_list, var_str ("expr")))
      call parse_node_append_sub (range%pn_expr, range%pn_term)
      call parse_node_freeze_branch (range%pn_expr)
    end subroutine range_create_value_node
```

Compile, given an environment.

⟨*Commands: range: TBP*⟩+≡
```
    procedure :: compile => range_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine range_compile (range, global)
      class(range_t), intent(inout) :: range
      type(rt_data_t), intent(in), target :: global
      if (associated (range%pn_beg)) then
         call eval_tree_init_expr &
                (range%expr_beg, range%pn_beg, global%var_list)
         if (associated (range%pn_end)) then
            call eval_tree_init_expr &
                   (range%expr_end, range%pn_end, global%var_list)
            if (associated (range%pn_step)) then
               call eval_tree_init_expr &
                      (range%expr_step, range%pn_step, global%var_list)
            end if
         end if
      end if
    end subroutine range_compile
```

Evaluate: compute the actual bounds and parameters that determine the values
that we can iterate.

This is implementation-specific.

⟨*Commands: range: TBP*⟩+≡

```
procedure (range_evaluate), deferred :: evaluate
```

⟨*Commands: interfaces*⟩+≡

```
abstract interface
   subroutine range_evaluate (range)
     import
     class(range_t), intent(inout) :: range
   end subroutine range_evaluate
end interface
```

The version for an integer variable. If the step is subtractive, we invert the
sign and treat it as an additive step. For a multiplicative step, the step must
be greater than one, and the initial and final values must be of same sign and
strictly ordered. Analogously for a division step.

⟨*Commands: range int: TBP*⟩+≡

```
procedure :: evaluate => range_int_evaluate
```

⟨*Commands: procedures*⟩+≡

```
subroutine range_int_evaluate (range)
  class(range_int_t), intent(inout) :: range
  integer :: ival
  if (associated (range%pn_beg)) then
     call eval_tree_evaluate (range%expr_beg)
     if (eval_tree_result_is_known (range%expr_beg)) then
        range%i_beg = eval_tree_get_int (range%expr_beg)
     else
        call range%write ()
        call msg_fatal &
             ("Range expression: initial value evaluates to unknown")
     end if
     if (associated (range%pn_end)) then
        call eval_tree_evaluate (range%expr_end)
        if (eval_tree_result_is_known (range%expr_end)) then
           range%i_end = eval_tree_get_int (range%expr_end)
           if (associated (range%pn_step)) then
              call eval_tree_evaluate (range%expr_step)
              if (eval_tree_result_is_known (range%expr_step)) then
                 range%i_step = eval_tree_get_int (range%expr_step)
                 select case (range%step_mode)
                 case (STEP_SUB);  range%i_step = - range%i_step
                 end select
              else
                 call range%write ()
                 call msg_fatal &
                      ("Range expression: step value evaluates to unknown")
              end if
           else
              range%i_step = 1
           end if
        else
```

```
         call range%write ()
         call msg_fatal &
              ("Range expression: final value evaluates to unknown")
      end if
else
   range%i_end = range%i_beg
   range%i_step = 1
end if
select case (range%step_mode)
case (STEP_NONE)
   range%n_step = 1
case (STEP_ADD, STEP_SUB)
   if (range%i_step /= 0) then
      if (range%i_beg == range%i_end) then
         range%n_step = 1
      else if (sign (1, range%i_end - range%i_beg) &
           == sign (1, range%i_step)) then
         range%n_step = (range%i_end - range%i_beg) / range%i_step + 1
      else
         range%n_step = 0
      end if
   else
      call msg_fatal ("range evaluation (add): step value is zero")
   end if
case (STEP_MUL)
   if (range%i_step > 1) then
      if (range%i_beg == range%i_end) then
         range%n_step = 1
      else if (range%i_beg == 0) then
         call msg_fatal ("range evaluation (mul): initial value is zero")
      else if (sign (1, range%i_beg) == sign (1, range%i_end) &
           .and. abs (range%i_beg) < abs (range%i_end)) then
         range%n_step = 0
         ival = range%i_beg
         do while (abs (ival) <= abs (range%i_end))
             range%n_step = range%n_step + 1
             ival = ival * range%i_step
         end do
      else
         range%n_step = 0
      end if
   else
      call msg_fatal &
           ("range evaluation (mult): step value is one or less")
   end if
case (STEP_DIV)
   if (range%i_step > 1) then
      if (range%i_beg == range%i_end) then
         range%n_step = 1
      else if (sign (1, range%i_beg) == sign (1, range%i_end) &
           .and. abs (range%i_beg) > abs (range%i_end)) then
         range%n_step = 0
         ival = range%i_beg
         do while (abs (ival) >= abs (range%i_end))
```

```
                        range%n_step = range%n_step + 1
                        if (ival == 0)  exit
                        ival = ival / range%i_step
                     end do
                 else
                     range%n_step = 0
                 end if
              else
                 call msg_fatal &
                      ("range evaluation (div): step value is one or less")
              end if
           case (STEP_COMP_ADD)
              call msg_fatal ("range evaluation: &
                   &step mode /+/ not allowed for integer variable")
           case (STEP_COMP_MUL)
              call msg_fatal ("range evaluation: &
                   &step mode /*/ not allowed for integer variable")
           case default
              call range%write ()
              call msg_bug ("range evaluation: step mode not implemented")
           end select
        end if
      end subroutine range_int_evaluate
```

The version for a real variable.

⟨Commands: range real: TBP⟩+≡
```
      procedure :: evaluate => range_real_evaluate
```

⟨Commands: procedures⟩+≡
```
      subroutine range_real_evaluate (range)
        class(range_real_t), intent(inout) :: range
        if (associated (range%pn_beg)) then
           call eval_tree_evaluate (range%expr_beg)
           if (eval_tree_result_is_known (range%expr_beg)) then
              range%r_beg = eval_tree_get_real (range%expr_beg)
           else
              call range%write ()
              call msg_fatal &
                   ("Range expression: initial value evaluates to unknown")
           end if
           if (associated (range%pn_end)) then
              call eval_tree_evaluate (range%expr_end)
              if (eval_tree_result_is_known (range%expr_end)) then
                 range%r_end = eval_tree_get_real (range%expr_end)
                 if (associated (range%pn_step)) then
                    if (eval_tree_result_is_known (range%expr_step)) then
                       select case (range%step_mode)
                       case (STEP_ADD, STEP_SUB, STEP_MUL, STEP_DIV)
                          call eval_tree_evaluate (range%expr_step)
                          range%r_step = eval_tree_get_real (range%expr_step)
                          select case (range%step_mode)
                          case (STEP_SUB);  range%r_step = - range%r_step
                          end select
                       case (STEP_COMP_ADD, STEP_COMP_MUL)
```

```
                  range%n_step = &
                       max (eval_tree_get_int (range%expr_step), 0)
               end select
            else
               call range%write ()
               call msg_fatal &
                    ("Range expression: step value evaluates to unknown")
            end if
         else
            call range%write ()
            call msg_fatal &
                 ("Range expression (real): step value must be provided")
         end if
      else
         call range%write ()
         call msg_fatal &
              ("Range expression: final value evaluates to unknown")
      end if
   else
      range%r_end = range%r_beg
      range%r_step = 1
   end if
   select case (range%step_mode)
   case (STEP_NONE)
      range%n_step = 1
   case (STEP_ADD, STEP_SUB)
      if (range%r_step /= 0) then
         if (sign (1._default, range%r_end - range%r_beg) &
              == sign (1._default, range%r_step)) then
            range%n_step = &
                 nint ((range%r_end - range%r_beg) / range%r_step + 1)
         else
            range%n_step = 0
         end if
      else
         call msg_fatal ("range evaluation (add): step value is zero")
      end if
   case (STEP_MUL)
      if (range%r_step > 1) then
         if (range%r_beg == 0 .or. range%r_end == 0) then
            call msg_fatal ("range evaluation (mul): bound is zero")
         else if (sign (1._default, range%r_beg) &
              == sign (1._default, range%r_end) &
              .and. abs (range%r_beg) <= abs (range%r_end)) then
            range%lr_beg = log (abs (range%r_beg))
            range%lr_end = log (abs (range%r_end))
            range%lr_step = log (range%r_step)
            range%n_step = nint &
                 (abs ((range%lr_end - range%lr_beg) / range%lr_step) + 1)
         else
            range%n_step = 0
         end if
      else
         call msg_fatal &
```

```
                            ("range evaluation (mult): step value is one or less")
                 end if
            case (STEP_DIV)
               if (range%r_step > 1) then
                  if (range%r_beg == 0 .or. range%r_end == 0) then
                     call msg_fatal ("range evaluation (div): bound is zero")
                  else if (sign (1._default, range%r_beg) &
                        == sign (1._default, range%r_end) &
                        .and. abs (range%r_beg) >= abs (range%r_end)) then
                     range%lr_beg = log (abs (range%r_beg))
                     range%lr_end = log (abs (range%r_end))
                     range%lr_step = -log (range%r_step)
                     range%n_step = nint &
                           (abs ((range%lr_end - range%lr_beg) / range%lr_step) + 1)
                  else
                     range%n_step = 0
                  end if
               else
                  call msg_fatal &
                        ("range evaluation (mult): step value is one or less")
               end if
            case (STEP_COMP_ADD)
               ! Number of steps already known
            case (STEP_COMP_MUL)
               ! Number of steps already known
               if (range%r_beg == 0 .or. range%r_end == 0) then
                  call msg_fatal ("range evaluation (mul): bound is zero")
               else if (sign (1._default, range%r_beg) &
                     == sign (1._default, range%r_end)) then
                  range%lr_beg = log (abs (range%r_beg))
                  range%lr_end = log (abs (range%r_end))
               else
                  range%n_step = 0
               end if
            case default
               call range%write ()
               call msg_bug ("range evaluation: step mode not implemented")
            end select
         end if
    end subroutine range_real_evaluate
```

Return the number of iterations:

⟨*Commands: range: TBP*⟩+≡
```
    procedure :: get_n_iterations => range_get_n_iterations
```

⟨*Commands: procedures*⟩+≡
```
    function range_get_n_iterations (range) result (n)
      class(range_t), intent(in) :: range
      integer :: n
      n = range%n_step
    end function range_get_n_iterations
```

Compute the value for iteration i and store it in the embedded token.

⟨*Commands: range: TBP*⟩+≡

```
        procedure (range_set_value), deferred :: set_value
```

⟨*Commands: interfaces*⟩+≡
```
    abstract interface
       subroutine range_set_value (range, i)
         import
         class(range_t), intent(inout) :: range
         integer, intent(in) :: i
       end subroutine range_set_value
    end interface
```

In the integer case, we compute the value directly for additive step. For multiplicative step, we perform a loop in the same way as above, where the number of iteration was determined.

⟨*Commands: range int: TBP*⟩+≡
```
    procedure :: set_value => range_int_set_value
```

⟨*Commands: procedures*⟩+≡
```
    subroutine range_int_set_value (range, i)
      class(range_int_t), intent(inout) :: range
      integer, intent(in) :: i
      integer :: k, ival
      select case (range%step_mode)
      case (STEP_NONE)
         ival = range%i_beg
      case (STEP_ADD, STEP_SUB)
         ival = range%i_beg + (i - 1) * range%i_step
      case (STEP_MUL)
         ival = range%i_beg
         do k = 1, i - 1
            ival = ival * range%i_step
         end do
      case (STEP_DIV)
         ival = range%i_beg
         do k = 1, i - 1
            ival = ival / range%i_step
         end do
      case default
         call range%write ()
         call msg_bug ("range iteration: step mode not implemented")
      end select
      call parse_node_set_value (range%pn_literal, ival = ival)
    end subroutine range_int_set_value
```

In the integer case, we compute the value directly for additive step. For multiplicative step, we perform a loop in the same way as above, where the number of iteration was determined.

⟨*Commands: range real: TBP*⟩+≡
```
    procedure :: set_value => range_real_set_value
```

⟨*Commands: procedures*⟩+≡
```
    subroutine range_real_set_value (range, i)
      class(range_real_t), intent(inout) :: range
      integer, intent(in) :: i
```

```
   real(default) :: rval, x
   select case (range%step_mode)
   case (STEP_NONE)
      rval = range%r_beg
   case (STEP_ADD, STEP_SUB, STEP_COMP_ADD)
      if (range%n_step > 1) then
         x = real (i - 1, default) / (range%n_step - 1)
      else
         x = 1._default / 2
      end if
      rval = x * range%r_end + (1 - x) * range%r_beg
   case (STEP_MUL, STEP_DIV, STEP_COMP_MUL)
      if (range%n_step > 1) then
         x = real (i - 1, default) / (range%n_step - 1)
      else
         x = 1._default / 2
      end if
      rval = sign &
            (exp (x * range%lr_end + (1 - x) * range%lr_beg), range%r_beg)
   case default
      call range%write ()
      call msg_bug ("range iteration: step mode not implemented")
   end select
   call parse_node_set_value (range%pn_literal, rval = rval)
 end subroutine range_real_set_value
```

Scan over parameters and other objects

The scan command allocates a new parse node for the variable assignment (the
lhs). The rhs of this parse node is assigned from the available rhs expressions
in the scan list, one at a time, so the compiled parse node can be prepended to
the scan body.

Note: for the integer/real range array, the obvious implementation as a
polymorphic array is suspended because in gfortran 4.7, polymorphic arrays are
apparently broken.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_scan_t
     private
     type(string_t) :: name
     integer :: n_values = 0
     type(parse_node_p), dimension(:), allocatable :: scan_cmd
     !!! !!! gfortran 4.7.x memory corruption
     !!!  class(range_t), dimension(:), allocatable :: range
     type(range_int_t), dimension(:), allocatable :: range_int
     type(range_real_t), dimension(:), allocatable :: range_real
   contains
   ⟨Commands: cmd scan: TBP⟩
   end type cmd_scan_t
```

Finalizer.

The auxiliary parse nodes that we have constructed have to be treated carefully: the embedded pointers all point to persistent objects somewhere else and should not be finalized, so we should not call the finalizer recursively.

⟨*Commands: cmd scan: TBP*⟩≡
```
procedure :: final => cmd_scan_final
```

⟨*Commands: procedures*⟩+≡
```
recursive subroutine cmd_scan_final (cmd)
  class(cmd_scan_t), intent(inout) :: cmd
  type(parse_node_t), pointer :: pn_var_single, pn_decl_single
  type(string_t) :: key
  integer :: i
  if (allocated (cmd%scan_cmd)) then
     do i = 1, size (cmd%scan_cmd)
        pn_var_single => parse_node_get_sub_ptr (cmd%scan_cmd(i)%ptr)
        key = parse_node_get_rule_key (pn_var_single)
        select case (char (key))
        case ("scan_string_decl", "scan_log_decl")
           pn_decl_single => parse_node_get_sub_ptr (pn_var_single, 2)
           call parse_node_final (pn_decl_single, recursive=.false.)
           deallocate (pn_decl_single)
        end select
        call parse_node_final (pn_var_single, recursive=.false.)
        deallocate (pn_var_single)
     end do
     deallocate (cmd%scan_cmd)
  end if
  !!! !!! gfortran 4.7.x memory corruption
  !!!  if (allocated (cmd%range)) then
  !!!     do i = 1, size (cmd%range)
  !!!        call cmd%range(i)%final ()
  !!!     end do
  !!!  end if
  if (allocated (cmd%range_int)) then
     do i = 1, size (cmd%range_int)
        call cmd%range_int(i)%final ()
     end do
  end if
  if (allocated (cmd%range_real)) then
     do i = 1, size (cmd%range_real)
        call cmd%range_real(i)%final ()
     end do
  end if
end subroutine cmd_scan_final
```

Output.

⟨*Commands: cmd scan: TBP*⟩+≡
```
procedure :: write => cmd_scan_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_scan_write (cmd, unit, indent)
  class(cmd_scan_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
```

```
      u = output_unit (unit);  if (u < 0)  return
      call write_indent (u, indent)
      write (u, "(1x,A,1x,A,1x,'(',I0,')')")  "scan:", char (cmd%name), &
           cmd%n_values
   end subroutine cmd_scan_write
```

Compile the scan command. We construct a new parse node that implements the variable assignment for a single element on the rhs, instead of the whole list that we get from the original parse tree. By simply copying the node, we copy all pointers and inherit the targets from the original. During execution, we should replace the rhs by the stored rhs pointers (the list elements), one by one, then (re)compile the redefined node.

⟨*Commands: cmd scan: TBP*⟩+≡
```
   procedure :: compile => cmd_scan_compile
```

⟨*Commands: procedures*⟩+≡
```
   recursive subroutine cmd_scan_compile (cmd, global)
     class(cmd_scan_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), pointer :: pn_var, pn_body, pn_body_first
     type(parse_node_t), pointer :: pn_decl, pn_name
     type(parse_node_t), pointer :: pn_arg, pn_scan_cmd, pn_rhs
     type(parse_node_t), pointer :: pn_decl_single, pn_var_single
     type(syntax_rule_t), pointer :: var_rule_decl, var_rule
     type(string_t) :: key
     integer :: var_type
     integer :: i
     logical, parameter :: debug = .false.
     if (debug) then
        print *, "compile scan"
        call parse_node_write_rec (cmd%pn)
     end if
     pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
     pn_body => parse_node_get_next_ptr (pn_var)
     if (associated (pn_body)) then
        pn_body_first => parse_node_get_sub_ptr (pn_body)
     else
        pn_body_first => null ()
     end if
     key = parse_node_get_rule_key (pn_var)
     select case (char (key))
     case ("scan_num")
        pn_name => parse_node_get_sub_ptr (pn_var)
        cmd%name = parse_node_get_string (pn_name)
        var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_num"))
        pn_arg => parse_node_get_next_ptr (pn_name, 2)
     case ("scan_int")
        pn_name => parse_node_get_sub_ptr (pn_var, 2)
        cmd%name = parse_node_get_string (pn_name)
        var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_int"))
        pn_arg => parse_node_get_next_ptr (pn_name, 2)
     case ("scan_real")
        pn_name => parse_node_get_sub_ptr (pn_var, 2)
        cmd%name = parse_node_get_string (pn_name)
```

3111

```
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_real"))
      pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_complex")
      pn_name => parse_node_get_sub_ptr (pn_var, 2)
      cmd%name = parse_node_get_string (pn_name)
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str("cmd_complex"))
      pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_alias")
      pn_name => parse_node_get_sub_ptr (pn_var, 2)
      cmd%name = parse_node_get_string (pn_name)
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_alias"))
      pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_string_decl")
      pn_decl => parse_node_get_sub_ptr (pn_var, 2)
      pn_name => parse_node_get_sub_ptr (pn_decl, 2)
      cmd%name = parse_node_get_string (pn_name)
      var_rule_decl => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_string"))
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_string_decl"))
      pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_log_decl")
      pn_decl => parse_node_get_sub_ptr (pn_var, 2)
      pn_name => parse_node_get_sub_ptr (pn_decl, 2)
      cmd%name = parse_node_get_string (pn_name)
      var_rule_decl => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_log"))
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_log_decl"))
      pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_cuts")
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_cuts"))
      cmd%name = "cuts"
      pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_weight")
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_weight"))
      cmd%name = "weight"
      pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_scale")
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_scale"))
      cmd%name = "scale"
      pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_ren_scale")
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_ren_scale"))
      cmd%name = "renormalization_scale"
      pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_fac_scale")
      var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
            var_str ("cmd_fac_scale"))
      cmd%name = "factorization_scale"
```

```
         pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_selection")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
         var_str ("cmd_selection"))
   cmd%name = "selection"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_reweight")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
         var_str ("cmd_reweight"))
   cmd%name = "reweight"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_analysis")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
         var_str ("cmd_analysis"))
   cmd%name = "analysis"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_model")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
         var_str ("cmd_model"))
   cmd%name = "model"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_library")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
         var_str ("cmd_library"))
   cmd%name = "library"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case default
   call msg_bug ("scan: case '" // char (key) // "' not implemented")
end select
if (associated (pn_arg)) then
   cmd%n_values = parse_node_get_n_sub (pn_arg)
end if
allocate (cmd%scan_cmd (cmd%n_values))
select case (char (key))
case ("scan_num")
   var_type = var_list_get_type (global%var_list, cmd%name)
   select case (var_type)
   case (V_INT)
      !!! !!! gfortran 4.7.x memory corruption
      !!!  allocate (range_int_t :: cmd%range (cmd%n_values))
      allocate (cmd%range_int (cmd%n_values))
   case (V_REAL)
      !!! !!! gfortran 4.7.x memory corruption
      !!!  allocate (range_real_t :: cmd%range (cmd%n_values))
      allocate (cmd%range_real (cmd%n_values))
   case (V_CMPLX)
      call msg_fatal ("scan over complex variable not implemented")
   case (V_NONE)
      call msg_fatal ("scan: variable '" // char (cmd%name) //"' undefined")
   case default
      call msg_bug ("scan: impossible variable type")
   end select
case ("scan_int")
   !!! !!! gfortran 4.7.x memory corruption
```

```
!!!  allocate (range_int_t :: cmd%range (cmd%n_values))
   allocate (cmd%range_int (cmd%n_values))
case ("scan_real")
   !!! !!! gfortran 4.7.x memory corruption
   !!!  allocate (range_real_t :: cmd%range (cmd%n_values))
   allocate (cmd%range_real (cmd%n_values))
case ("scan_complex")
   call msg_fatal ("scan over complex variable not implemented")
end select
i = 1
if (associated (pn_arg)) then
   pn_rhs => parse_node_get_sub_ptr (pn_arg)
else
   pn_rhs => null ()
end if
do while (associated (pn_rhs))
   allocate (pn_scan_cmd)
   call parse_node_create_branch (pn_scan_cmd, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("command_list")))
   allocate (pn_var_single)
   pn_var_single = pn_var
   call parse_node_replace_rule (pn_var_single, var_rule)
   select case (char (key))
   case ("scan_num", "scan_int", "scan_real", &
         "scan_complex", "scan_alias", &
         "scan_cuts", "scan_weight", &
         "scan_scale", "scan_ren_scale", "scan_fac_scale", &
         "scan_selection", "scan_reweight", "scan_analysis", &
         "scan_model", "scan_library")
      if (allocated (cmd%range_int)) then
         call cmd%range_int(i)%init (pn_rhs)
         !!! !!! gfortran 4.7.x memory corruption
         !!!  call cmd%range_int(i)%compile (global)
         call parse_node_replace_last_sub &
              (pn_var_single, cmd%range_int(i)%pn_expr)
      else if (allocated (cmd%range_real)) then
         call cmd%range_real(i)%init (pn_rhs)
         !!! !!! gfortran 4.7.x memory corruption
         !!!  call cmd%range_real(i)%compile (global)
         call parse_node_replace_last_sub &
              (pn_var_single, cmd%range_real(i)%pn_expr)
      else
         call parse_node_replace_last_sub (pn_var_single, pn_rhs)
      end if
   case ("scan_string_decl", "scan_log_decl")
      allocate (pn_decl_single)
      pn_decl_single = pn_decl
      call parse_node_replace_rule (pn_decl_single, var_rule_decl)
      call parse_node_replace_last_sub (pn_decl_single, pn_rhs)
      call parse_node_freeze_branch (pn_decl_single)
      call parse_node_replace_last_sub (pn_var_single, pn_decl_single)
   case default
      call msg_bug ("scan: case '" // char (key)  &
            // "' broken")
```

```
              end select
              call parse_node_freeze_branch (pn_var_single)
              call parse_node_append_sub (pn_scan_cmd, pn_var_single)
              call parse_node_append_sub (pn_scan_cmd, pn_body_first)
              call parse_node_freeze_branch (pn_scan_cmd)
              cmd%scan_cmd(i)%ptr => pn_scan_cmd
              i = i + 1
              pn_rhs => parse_node_get_next_ptr (pn_rhs)
           end do
           if (debug) then
              do i = 1, cmd%n_values
                 print *, "scan command ", i
                 call parse_node_write_rec (cmd%scan_cmd(i)%ptr)
                 if (allocated (cmd%range_int))  call cmd%range_int(i)%write ()
                 if (allocated (cmd%range_real))  call cmd%range_real(i)%write ()
              end do
              print *, "original"
              call parse_node_write_rec (cmd%pn)
           end if
         end subroutine cmd_scan_compile
```

Execute the loop for all values in the step list. We use the parse trees with single variable assignment that we have stored, to iteratively create a local environment, execute the stored commands, and destroy it again. When we encounter a range object, we execute the commands for each value that this object provides. Computing this value has the side effect of modifying the rhs of the variable assignment that heads the local command list, directly in the local parse tree.

⟨*Commands: cmd scan: TBP*⟩+≡
```
       procedure :: execute => cmd_scan_execute
```
⟨*Commands: procedures*⟩+≡
```
       recursive subroutine cmd_scan_execute (cmd, global)
         class(cmd_scan_t), intent(inout) :: cmd
         type(rt_data_t), intent(inout), target :: global
         type(rt_data_t), allocatable :: local
         integer :: i, j
         do i = 1, cmd%n_values
            if (allocated (cmd%range_int)) then
               call cmd%range_int(i)%compile (global)
               call cmd%range_int(i)%evaluate ()
               do j = 1, cmd%range_int(i)%get_n_iterations ()
                  call cmd%range_int(i)%set_value (j)
                  allocate (local)
                  call build_alt_setup (local, global, cmd%scan_cmd(i)%ptr)
                  deallocate (local)
               end do
            else if (allocated (cmd%range_real)) then
               call cmd%range_real(i)%compile (global)
               call cmd%range_real(i)%evaluate ()
               do j = 1, cmd%range_real(i)%get_n_iterations ()
                  call cmd%range_real(i)%set_value (j)
                  allocate (local)
                  call build_alt_setup (local, global, cmd%scan_cmd(i)%ptr)
```

```
         deallocate (local)
      end do
   else
      allocate (local)
      call build_alt_setup (local, global, cmd%scan_cmd(i)%ptr)
      deallocate (local)
   end if
  end do
end subroutine cmd_scan_execute
```

Conditionals

Conditionals are implemented as a list that is compiled and evaluated recursively; this allows for a straightforward representation of `else if` constructs. A `cmd_if_t` object can hold either an `else_if` clause which is another object of this type, or an `else_body`, but not both.

If- or else-bodies are no scoping units, so all data remain global and no copy-in copy-out is needed.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_if_t
     private
     type(parse_node_t), pointer :: pn_if_lexpr => null ()
     type(command_list_t), pointer :: if_body => null ()
     type(cmd_if_t), dimension(:), pointer :: elsif_cmd => null ()
     type(command_list_t), pointer :: else_body => null ()
    contains
    ⟨Commands: cmd if: TBP⟩
   end type cmd_if_t
```

Finalizer. There are no local options, therefore we can simply override the default finalizer.

⟨*Commands: cmd if: TBP*⟩≡
```
   procedure :: final => cmd_if_final
```

⟨*Commands: procedures*⟩+≡
```
   recursive subroutine cmd_if_final (cmd)
     class(cmd_if_t), intent(inout) :: cmd
     integer :: i
     if (associated (cmd%if_body)) then
        call command_list_final (cmd%if_body)
        deallocate (cmd%if_body)
     end if
     if (associated (cmd%elsif_cmd)) then
        do i = 1, size (cmd%elsif_cmd)
           call cmd_if_final (cmd%elsif_cmd(i))
        end do
        deallocate (cmd%elsif_cmd)
     end if
     if (associated (cmd%else_body)) then
        call command_list_final (cmd%else_body)
        deallocate (cmd%else_body)
     end if
```

```
      end subroutine cmd_if_final
```

Output. Recursively write the command lists.

⟨*Commands: cmd if: TBP*⟩+≡
```
    procedure :: write => cmd_if_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_if_write (cmd, unit, indent)
      class(cmd_if_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u, ind, i
      u = output_unit (unit);  if (u < 0)  return
      ind = 0;  if (present (indent))  ind = indent
      call write_indent (u, indent)
      write (u, "(A)")  "if <expr> then"
      if (associated (cmd%if_body)) then
         call cmd%if_body%write (unit, ind + 1)
      end if
      if (associated (cmd%elsif_cmd)) then
         do i = 1, size (cmd%elsif_cmd)
            call write_indent (u, indent)
            write (u, "(A)")  "elsif <expr> then"
            if (associated (cmd%elsif_cmd(i)%if_body)) then
               call cmd%elsif_cmd(i)%if_body%write (unit, ind + 1)
            end if
         end do
      end if
      if (associated (cmd%else_body)) then
         call write_indent (u, indent)
         write (u, "(A)")  "else"
         call cmd%else_body%write (unit, ind + 1)
      end if
    end subroutine cmd_if_write
```

Compile the conditional.

⟨*Commands: cmd if: TBP*⟩+≡
```
    procedure :: compile => cmd_if_compile
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine cmd_if_compile (cmd, global)
      class(cmd_if_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_lexpr, pn_body
      type(parse_node_t), pointer :: pn_elsif_clauses, pn_cmd_elsif
      type(parse_node_t), pointer :: pn_else_clause, pn_cmd_else
      integer :: i, n_elsif
      pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 2)
      cmd%pn_if_lexpr => pn_lexpr
      pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
      select case (char (parse_node_get_rule_key (pn_body)))
      case ("command_list")
         allocate (cmd%if_body)
         call cmd%if_body%compile (pn_body, global)
         pn_elsif_clauses => parse_node_get_next_ptr (pn_body)
```

```
         case default
            pn_elsif_clauses => pn_body
         end select
         select case (char (parse_node_get_rule_key (pn_elsif_clauses)))
         case ("elsif_clauses")
            n_elsif = parse_node_get_n_sub (pn_elsif_clauses)
            allocate (cmd%elsif_cmd (n_elsif))
            pn_cmd_elsif => parse_node_get_sub_ptr (pn_elsif_clauses)
            do i = 1, n_elsif
               pn_lexpr => parse_node_get_sub_ptr (pn_cmd_elsif, 2)
               cmd%elsif_cmd(i)%pn_if_lexpr => pn_lexpr
               pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
               if (associated (pn_body)) then
                  allocate (cmd%elsif_cmd(i)%if_body)
                  call cmd%elsif_cmd(i)%if_body%compile (pn_body, global)
               end if
               pn_cmd_elsif => parse_node_get_next_ptr (pn_cmd_elsif)
            end do
            pn_else_clause => parse_node_get_next_ptr (pn_elsif_clauses)
         case default
            pn_else_clause => pn_elsif_clauses
         end select
         select case (char (parse_node_get_rule_key (pn_else_clause)))
         case ("else_clause")
            pn_cmd_else => parse_node_get_sub_ptr (pn_else_clause)
            pn_body => parse_node_get_sub_ptr (pn_cmd_else, 2)
            if (associated (pn_body)) then
               allocate (cmd%else_body)
               call cmd%else_body%compile (pn_body, global)
            end if
         end select
    end subroutine cmd_if_compile
```

(Recursively) execute the condition. Context remains global in all cases.

⟨*Commands: cmd if: TBP*⟩+≡
```
    procedure :: execute => cmd_if_execute
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine cmd_if_execute (cmd, global)
      class(cmd_if_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      logical :: lval, is_known
      integer :: i
      lval = eval_log (cmd%pn_if_lexpr, global%var_list, is_known=is_known)
      if (is_known) then
         if (lval) then
            if (associated (cmd%if_body)) then
               call cmd%if_body%execute (global)
            end if
            return
         end if
      else
         call error_undecided ()
         return
```

```
      end if
      if (associated (cmd%elsif_cmd)) then
         SCAN_ELSIF: do i = 1, size (cmd%elsif_cmd)
            lval = eval_log (cmd%elsif_cmd(i)%pn_if_lexpr, global%var_list, &
                  is_known=is_known)
            if (is_known) then
               if (lval) then
                  if (associated (cmd%elsif_cmd(i)%if_body)) then
                     call cmd%elsif_cmd(i)%if_body%execute (global)
                  end if
                  return
               end if
            else
               call error_undecided ()
               return
            end if
         end do SCAN_ELSIF
      end if
      if (associated (cmd%else_body)) then
         call cmd%else_body%execute (global)
      end if
   contains
     subroutine error_undecided ()
       call msg_error ("Undefined result of cmditional expression: " &
            // "neither branch will be executed")
     end subroutine error_undecided
   end subroutine cmd_if_execute
```

Include another command-list file

The include command allocates a local parse tree. This must not be deleted before the command object itself is deleted, since pointers may point to subobjects of it.

⟨*Commands: types*⟩+≡
```
   type, extends (command_t) :: cmd_include_t
     private
     type(string_t) :: file
     type(command_list_t), pointer :: command_list => null ()
     type(parse_tree_t) :: parse_tree
    contains
    ⟨Commands: cmd include: TBP⟩
   end type cmd_include_t
```

Finalizer: delete the command list. No options, so we can simply override the default finalizer.

⟨*Commands: cmd include: TBP*⟩≡
```
   procedure :: final => cmd_include_final
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_include_final (cmd)
     class(cmd_include_t), intent(inout) :: cmd
     call parse_tree_final (cmd%parse_tree)
```

```
      if (associated (cmd%command_list)) then
         call cmd%command_list%final ()
         deallocate (cmd%command_list)
      end if
   end subroutine cmd_include_final
```

Write: display the command list as-is, if allocated.

⟨*Commands: cmd include: TBP*⟩+≡
```
   procedure :: write => cmd_include_write
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_include_write (cmd, unit, indent)
     class(cmd_include_t), intent(in) :: cmd
     integer, intent(in), optional :: unit, indent
     integer :: u, ind
     u = output_unit (unit)
     ind = 0;  if (present (indent))  ind = indent
     call write_indent (u, indent)
     write (u, "(A,A,A,A)")  "include ", ’"’, char (cmd%file), ’"’
     if (associated (cmd%command_list)) then
        call cmd%command_list%write (u, ind + 1)
     end if
   end subroutine cmd_include_write
```

Compile file contents: First parse the file, then immediately compile its contents.
Use the global data set.

⟨*Commands: cmd include: TBP*⟩+≡
```
   procedure :: compile => cmd_include_compile
```

⟨*Commands: procedures*⟩+≡
```
   subroutine cmd_include_compile (cmd, global)
     class(cmd_include_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), pointer :: pn_arg, pn_file
     type(string_t) :: file
     logical :: exist
     integer :: u
     type(stream_t), target :: stream
     type(lexer_t) :: lexer
     pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
     pn_file => parse_node_get_sub_ptr (pn_arg)
     file = parse_node_get_string (pn_file)
     inquire (file=char(file), exist=exist)
     if (exist) then
        cmd%file = file
     else
        cmd%file = global%os_data%whizard_cutspath // "/" // file
        inquire (file=char(cmd%file), exist=exist)
        if (.not. exist) then
           call msg_error ("Include file ’" // char (file) // "’ not found")
           return
        end if
     end if
     u = free_unit ()
```

```
    call lexer_init_cmd_list (lexer, global%lexer)
    call stream_init (stream, char (cmd%file))
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (cmd%parse_tree, syntax_cmd_list, lexer)
    call stream_final (stream)
    call lexer_final (lexer)
    close (u)
    allocate (cmd%command_list)
    call cmd%command_list%compile (parse_tree_get_root_ptr (cmd%parse_tree), &
        global)
  end subroutine cmd_include_compile
```

Execute file contents in the global context.

⟨*Commands: cmd include: TBP*⟩+≡
```
    procedure :: execute => cmd_include_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_include_execute (cmd, global)
      class(cmd_include_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      if (associated (cmd%command_list)) then
         call msg_message &
              ("Including Sindarin from '" // char (cmd%file) // "'")
         call cmd%command_list%execute (global)
         call msg_message &
              ("End of included '" // char (cmd%file) // "'")
      end if
    end subroutine cmd_include_execute
```

Quit command execution

The code is the return code of the whole program if it is terminated by this command.

⟨*Commands: types*⟩+≡
```
    type, extends (command_t) :: cmd_quit_t
      private
      logical :: has_code = .false.
      type(parse_node_t), pointer :: pn_code_expr => null ()
     contains
     ⟨Commands: cmd quit: TBP⟩
    end type cmd_quit_t
```

Output.

⟨*Commands: cmd quit: TBP*⟩≡
```
    procedure :: write => cmd_quit_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_quit_write (cmd, unit, indent)
      class(cmd_quit_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
      integer :: u
      u = output_unit (unit);  if (u < 0)  return
```

```
      call write_indent (u, indent)
      write (u, "(1x,A,L1)")  "quit: has_code = ", cmd%has_code
    end subroutine cmd_quit_write
```

Compile: allocate a `quit` object which serves as a placeholder.

⟨*Commands: cmd quit: TBP*⟩+≡
```
    procedure :: compile => cmd_quit_compile
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_quit_compile (cmd, global)
      class(cmd_quit_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(parse_node_t), pointer :: pn_arg
      pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
      if (associated (pn_arg)) then
         cmd%pn_code_expr => parse_node_get_sub_ptr (pn_arg)
         cmd%has_code = .true.
      end if
    end subroutine cmd_quit_compile
```

Execute: The quit command does not execute anything, it just stops command execution. This is achieved by setting quit flag and quit code in the global variable list. However, the return code, if present, is an expression which has to be evaluated.

⟨*Commands: cmd quit: TBP*⟩+≡
```
    procedure :: execute => cmd_quit_execute
```

⟨*Commands: procedures*⟩+≡
```
    subroutine cmd_quit_execute (cmd, global)
      class(cmd_quit_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      logical :: is_known
      if (cmd%has_code) then
         global%quit_code = eval_int (cmd%pn_code_expr, global%var_list, &
              is_known=is_known)
         if (.not. is_known) then
            call msg_error ("Undefined return code of quit/exit command")
         end if
      end if
      global%quit = .true.
    end subroutine cmd_quit_execute
```

23.2.2   The command list

The command list holds a list of commands and relevant global data.

⟨*Commands: public*⟩+≡
```
    public :: command_list_t
```

⟨*Commands: types*⟩+≡
```
    type :: command_list_t
       ! not private anymore as required by the whizard-c-interface
       class(command_t), pointer :: first => null ()
```

```
      class(command_t), pointer :: last => null ()
    contains
    ⟨Commands: command list: TBP⟩
  end type command_list_t
```

Output.

⟨*Commands: command list: TBP*⟩≡
```
    procedure :: write => command_list_write
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_list_write (cmd_list, unit, indent)
      class(command_list_t), intent(in) :: cmd_list
      integer, intent(in), optional :: unit, indent
      class(command_t), pointer :: cmd
      cmd => cmd_list%first
      do while (associated (cmd))
        call cmd%write (unit, indent)
        cmd => cmd%next
      end do
    end subroutine command_list_write
```

Append a new command to the list and free the original pointer.

⟨*Commands: command list: TBP*⟩+≡
```
    procedure :: append => command_list_append
```

⟨*Commands: procedures*⟩+≡
```
    subroutine command_list_append (cmd_list, command)
      class(command_list_t), intent(inout) :: cmd_list
      class(command_t), intent(inout), pointer :: command
      if (associated (cmd_list%last)) then
        cmd_list%last%next => command
      else
        cmd_list%first => command
      end if
      cmd_list%last => command
      command => null ()
    end subroutine command_list_append
```

Finalize.

⟨*Commands: command list: TBP*⟩+≡
```
    procedure :: final => command_list_final
```

⟨*Commands: procedures*⟩+≡
```
    recursive subroutine command_list_final (cmd_list)
      class(command_list_t), intent(inout) :: cmd_list
      class(command_t), pointer :: command
      do while (associated (cmd_list%first))
        command => cmd_list%first
        cmd_list%first => cmd_list%first%next
        call command%final ()
        deallocate (command)
      end do
      cmd_list%last => null ()
    end subroutine command_list_final
```

23.2.3 Compiling the parse tree

Transform a parse tree into a command list. Initialization is assumed to be done.

After each command, we set a breakpoint.

⟨*Commands: command list: TBP*⟩+≡

```
procedure :: compile => command_list_compile
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_list_compile (cmd_list, pn, global)
  class(command_list_t), intent(inout), target :: cmd_list
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_cmd
  class(command_t), pointer :: command
  integer :: i
  pn_cmd => parse_node_get_sub_ptr (pn)
  do i = 1, parse_node_get_n_sub (pn)
     call dispatch_command (command, pn_cmd)
     call command%compile (global)
     call cmd_list%append (command)
     call terminate_now_if_signal ()
     pn_cmd => parse_node_get_next_ptr (pn_cmd)
  end do
end subroutine command_list_compile
```

23.2.4 Executing the command list

Before executing a command we should execute its options (if any). After that, reset the options, i.e., remove temporary effects from the global state.

Also here, after each command we set a breakpoint.

⟨*Commands: command list: TBP*⟩+≡

```
procedure :: execute => command_list_execute
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_list_execute (cmd_list, global)
  class(command_list_t), intent(in) :: cmd_list
  type(rt_data_t), intent(inout), target :: global
  class(command_t), pointer :: command
  command => cmd_list%first
  COMMAND_COND: do while (associated (command))
     call command%execute_options (global)
     call command%execute (global)
     call command%reset_options (global)
     call terminate_now_if_signal ()
     if (global%quit)  exit COMMAND_COND
     command => command%next
  end do COMMAND_COND
end subroutine command_list_execute
```

23.2.5 Command list syntax

⟨*Commands: public*⟩+≡
```
    public :: syntax_cmd_list
```

⟨*Commands: variables*⟩≡
```
    type(syntax_t), target, save :: syntax_cmd_list
```

⟨*Commands: public*⟩+≡
```
    public :: syntax_cmd_list_init
```

⟨*Commands: procedures*⟩+≡
```
    subroutine syntax_cmd_list_init ()
      type(ifile_t) :: ifile
      call define_cmd_list_syntax (ifile)
      call syntax_init (syntax_cmd_list, ifile)
      call ifile_final (ifile)
    end subroutine syntax_cmd_list_init
```

⟨*Commands: public*⟩+≡
```
    public :: syntax_cmd_list_final
```

⟨*Commands: procedures*⟩+≡
```
    subroutine syntax_cmd_list_final ()
      call syntax_final (syntax_cmd_list)
    end subroutine syntax_cmd_list_final
```

⟨*Commands: public*⟩+≡
```
    public :: syntax_cmd_list_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine syntax_cmd_list_write (unit)
      integer, intent(in), optional :: unit
      call syntax_write (syntax_cmd_list, unit)
    end subroutine syntax_cmd_list_write
```

⟨*Commands: procedures*⟩+≡
```
    subroutine define_cmd_list_syntax (ifile)
      type(ifile_t), intent(inout) :: ifile
      call ifile_append (ifile, "SEQ command_list = command*")
      call ifile_append (ifile, "ALT command = " &
           // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
           // "cmd_var | cmd_slha | " &
           // "cmd_show | cmd_clear | " &
           // "cmd_expect | " &
           // "cmd_cuts | cmd_scale | cmd_fac_scale | cmd_ren_scale | " &
           // "cmd_weight | cmd_selection | cmd_reweight | " &
           // "cmd_beams | cmd_beams_pol_density | cmd_beams_pol_fraction | " &
           // "cmd_beams_momentum | cmd_beams_theta | cmd_beams_phi | " &
           // "cmd_integrate | " &
           // "cmd_observable | cmd_histogram | cmd_plot | cmd_graph | " &
           // "cmd_record | " &
           // "cmd_analysis | cmd_alt_setup | " &
           // "cmd_unstable | cmd_stable | cmd_simulate | cmd_rescan | " &
           // "cmd_process | cmd_compile | cmd_exec | " &
```

```
        // "cmd_scan | cmd_if | cmd_include | cmd_quit | " &
        // "cmd_polarized | cmd_unpolarized | " &
        // "cmd_open_out | cmd_close_out | cmd_printf | " &
        // "cmd_write_analysis | cmd_compile_analysis")
call ifile_append (ifile, "GRO options = '{' local_command_list '}'")
call ifile_append (ifile, "SEQ local_command_list = local_command*")
call ifile_append (ifile, "ALT local_command = " &
        // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
        // "cmd_var | cmd_slha | " &
        // "cmd_show | " &
        // "cmd_expect | " &
        // "cmd_cuts | cmd_scale | cmd_fac_scale | cmd_ren_scale | " &
        // "cmd_weight | cmd_selection | cmd_reweight | " &
        // "cmd_beams | cmd_beams_pol_density | cmd_beams_pol_fraction | " &
        // "cmd_beams_momentum | cmd_beams_theta | cmd_beams_phi | " &
        // "cmd_observable | cmd_histogram | cmd_plot | cmd_graph | " &
        // "cmd_clear | cmd_record | " &
        // "cmd_analysis | cmd_alt_setup | " &
        // "cmd_open_out | cmd_close_out | cmd_printf | " &
        // "cmd_write_analysis | cmd_compile_analysis")
call ifile_append (ifile, "SEQ cmd_model = model '=' model_name")
call ifile_append (ifile, "KEY model")
call ifile_append (ifile, "ALT model_name = model_id | string_literal")
call ifile_append (ifile, "IDE model_id")
call ifile_append (ifile, "SEQ cmd_library = library '=' lib_name")
call ifile_append (ifile, "KEY library")
call ifile_append (ifile, "ALT lib_name = lib_id | string_literal")
call ifile_append (ifile, "IDE lib_id")
call ifile_append (ifile, "ALT cmd_var = " &
        // "cmd_log_decl | cmd_log | " &
        // "cmd_int | cmd_real | cmd_complex | cmd_num | " &
        // "cmd_string_decl | cmd_string | cmd_alias | " &
        // "cmd_result")
call ifile_append (ifile, "SEQ cmd_log_decl = logical cmd_log")
call ifile_append (ifile, "SEQ cmd_log = '?' var_name '=' lexpr")
call ifile_append (ifile, "SEQ cmd_int = int var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_real = real var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_complex = complex var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_num = var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_string_decl = string cmd_string")
call ifile_append (ifile, "SEQ cmd_string = " &
        // "'$' var_name '=' sexpr") ! $
call ifile_append (ifile, "SEQ cmd_alias = alias var_name '=' cexpr")
call ifile_append (ifile, "SEQ cmd_result = result '=' expr")
call ifile_append (ifile, "SEQ cmd_slha = slha_action slha_arg options?")
call ifile_append (ifile, "ALT slha_action = " &
        // "read_slha | write_slha")
call ifile_append (ifile, "KEY read_slha")
call ifile_append (ifile, "KEY write_slha")
call ifile_append (ifile, "ARG slha_arg = ( string_literal )")
call ifile_append (ifile, "SEQ cmd_show = show show_arg options?")
call ifile_append (ifile, "KEY show")
call ifile_append (ifile, "ARG show_arg = ( showable* )")
call ifile_append (ifile, "ALT showable = " &
```

```
       // "model | library | beams | iterations | " &
       // "cuts | weight | logical | string | pdg | " &
       // "scale | factorization_scale | renormalization_scale | " &
       // "selection | reweight | analysis | " &
       // "stable | unstable | polarized | unpolarized | " &
       // "expect | intrinsic | int | real | complex | " &
       // "alias_var | string | results | result_var | " &
       // "log_var | string_var | var_name")
call ifile_append (ifile, "KEY results")
call ifile_append (ifile, "KEY intrinsic")
call ifile_append (ifile, "SEQ alias_var = alias var_name")
call ifile_append (ifile, "SEQ result_var = result_key result_arg?")
call ifile_append (ifile, "SEQ log_var = '?' var_name")
call ifile_append (ifile, "SEQ string_var = '$' var_name")  ! $
call ifile_append (ifile, "SEQ cmd_clear = clear clear_arg options?")
call ifile_append (ifile, "KEY clear")
call ifile_append (ifile, "ARG clear_arg = ( clearable* )")
call ifile_append (ifile, "ALT clearable = " &
       // "beams | iterations | " &
       // "cuts | weight | " &
       // "scale | factorization_scale | renormalization_scale | " &
       // "selection | reweight | analysis | " &
       // "unstable | polarized | " &
       // "expect | " &
       // "log_var | string_var | var_name")
call ifile_append (ifile, "SEQ cmd_expect = expect expect_arg options?")
call ifile_append (ifile, "KEY expect")
call ifile_append (ifile, "ARG expect_arg = ( lexpr )")
call ifile_append (ifile, "SEQ cmd_cuts = cuts '=' lexpr")
call ifile_append (ifile, "SEQ cmd_scale = scale '=' expr")
call ifile_append (ifile, "SEQ cmd_fac_scale = " &
       // "factorization_scale '=' expr")
call ifile_append (ifile, "SEQ cmd_ren_scale = " &
       // "renormalization_scale '=' expr")
call ifile_append (ifile, "SEQ cmd_weight = weight '=' expr")
call ifile_append (ifile, "SEQ cmd_selection = selection '=' lexpr")
call ifile_append (ifile, "SEQ cmd_reweight = reweight '=' expr")
call ifile_append (ifile, "KEY cuts")
call ifile_append (ifile, "KEY scale")
call ifile_append (ifile, "KEY factorization_scale")
call ifile_append (ifile, "KEY renormalization_scale")
call ifile_append (ifile, "KEY weight")
call ifile_append (ifile, "KEY selection")
call ifile_append (ifile, "KEY reweight")
call ifile_append (ifile, "SEQ cmd_process = process process_id '=' " &
       // "process_prt '=>' prt_state_list options?")
call ifile_append (ifile, "KEY process")
call ifile_append (ifile, "KEY '=>'")
call ifile_append (ifile, "LIS process_prt = cexpr+")
call ifile_append (ifile, "LIS prt_state_list = prt_state_sum+")
call ifile_append (ifile, "SEQ prt_state_sum = " &
       // "prt_state prt_state_addition*")
call ifile_append (ifile, "SEQ prt_state_addition = '+' prt_state")
call ifile_append (ifile, "ALT prt_state = grouped_prt_state_list | cexpr")
```

```
call ifile_append (ifile, "GRO grouped_prt_state_list = " &
     // "( prt_state_list )")
call ifile_append (ifile, "SEQ cmd_compile = compile_cmd options?")
call ifile_append (ifile, "SEQ compile_cmd = compile_clause compile_arg?")
call ifile_append (ifile, "SEQ compile_clause = compile exec_name_spec?")
call ifile_append (ifile, "KEY compile")
call ifile_append (ifile, "SEQ exec_name_spec = as exec_name")
call ifile_append (ifile, "KEY as")
call ifile_append (ifile, "ALT exec_name = exec_id | string_literal")
call ifile_append (ifile, "IDE exec_id")
call ifile_append (ifile, "ARG compile_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_exec = exec exec_arg")
call ifile_append (ifile, "KEY exec")
call ifile_append (ifile, "ARG exec_arg = ( sexpr )")
call ifile_append (ifile, "SEQ cmd_beams = beams '=' beam_def")
call ifile_append (ifile, "KEY beams")
call ifile_append (ifile, "SEQ beam_def = beam_spec strfun_seq*")
call ifile_append (ifile, "SEQ beam_spec = beam_list")
call ifile_append (ifile, "LIS beam_list = cexpr, cexpr?")
call ifile_append (ifile, "SEQ cmd_beams_pol_density = " &
     // "beams_pol_density '=' beams_pol_spec")
call ifile_append (ifile, "KEY beams_pol_density")
call ifile_append (ifile, "LIS beams_pol_spec = smatrix, smatrix?")
call ifile_append (ifile, "SEQ smatrix = '@' smatrix_arg")
! call ifile_append (ifile, "KEY '@'")       !!! Key already exists
call ifile_append (ifile, "ARG smatrix_arg = ( sentry* )")
call ifile_append (ifile, "SEQ sentry = expr extra_sentry*")
call ifile_append (ifile, "SEQ extra_sentry = ':' expr")
call ifile_append (ifile, "SEQ cmd_beams_pol_fraction = " &
     // "beams_pol_fraction '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_pol_fraction")
call ifile_append (ifile, "SEQ cmd_beams_momentum = " &
     // "beams_momentum '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_momentum")
call ifile_append (ifile, "SEQ cmd_beams_theta = " &
     // "beams_theta '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_theta")
call ifile_append (ifile, "SEQ cmd_beams_phi = " &
     // "beams_phi '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_phi")
call ifile_append (ifile, "LIS beams_par_spec = expr, expr?")
call ifile_append (ifile, "SEQ strfun_seq = '=>' strfun_pair")
call ifile_append (ifile, "LIS strfun_pair = strfun_def, strfun_def?")
call ifile_append (ifile, "SEQ strfun_def = strfun_id")
call ifile_append (ifile, "ALT strfun_id = " &
     // "none | lhapdf | lhapdf_photon | pdf_builtin | pdf_builtin_photon | " &
     // "isr | epa | ewa | circe1 | circe2 | energy_scan | " &
     // "beam_events | user_sf_spec")
call ifile_append (ifile, "KEY none")
call ifile_append (ifile, "KEY lhapdf")
call ifile_append (ifile, "KEY lhapdf_photon")
call ifile_append (ifile, "KEY pdf_builtin")
call ifile_append (ifile, "KEY pdf_builtin_photon")
call ifile_append (ifile, "KEY isr")
```

```
call ifile_append (ifile, "KEY epa")
call ifile_append (ifile, "KEY ewa")
call ifile_append (ifile, "KEY circe1")
call ifile_append (ifile, "KEY circe2")
call ifile_append (ifile, "KEY energy_scan")
call ifile_append (ifile, "KEY beam_events")
call ifile_append (ifile, "SEQ user_sf_spec = user_strfun user_arg")
call ifile_append (ifile, "KEY user_strfun")
call ifile_append (ifile, "SEQ cmd_integrate = " &
     // "integrate proc_arg options?")
call ifile_append (ifile, "KEY integrate")
call ifile_append (ifile, "ARG proc_arg = ( proc_id* )")
call ifile_append (ifile, "IDE proc_id")
call ifile_append (ifile, "SEQ cmd_iterations = " &
     // "iterations '=' iterations_list")
call ifile_append (ifile, "KEY iterations")
call ifile_append (ifile, "LIS iterations_list = iterations_spec+")
call ifile_append (ifile, "ALT iterations_spec = it_spec")
call ifile_append (ifile, "SEQ it_spec = expr calls_spec adapt_spec?")
call ifile_append (ifile, "SEQ calls_spec = ':' expr")
call ifile_append (ifile, "SEQ adapt_spec = ':' sexpr")
call ifile_append (ifile, "SEQ cmd_sample_format = " &
     // "sample_format '=' event_format_list")
call ifile_append (ifile, "KEY sample_format")
call ifile_append (ifile, "LIS event_format_list = event_format+")
call ifile_append (ifile, "IDE event_format")
call ifile_append (ifile, "SEQ cmd_observable = " &
     // "observable analysis_tag options?")
call ifile_append (ifile, "KEY observable")
call ifile_append (ifile, "SEQ cmd_histogram = " &
     // "histogram analysis_tag histogram_arg " &
     // "options?")
call ifile_append (ifile, "KEY histogram")
call ifile_append (ifile, "ARG histogram_arg = (expr, expr, expr?)")
call ifile_append (ifile, "SEQ cmd_plot = plot analysis_tag options?")
call ifile_append (ifile, "KEY plot")
call ifile_append (ifile, "SEQ cmd_graph = graph graph_term '=' graph_def")
call ifile_append (ifile, "KEY graph")
call ifile_append (ifile, "SEQ graph_term = analysis_tag options?")
call ifile_append (ifile, "SEQ graph_def = graph_term graph_append*")
call ifile_append (ifile, "SEQ graph_append = '&' graph_term")
call ifile_append (ifile, "SEQ cmd_analysis = analysis '=' lexpr")
call ifile_append (ifile, "KEY analysis")
call ifile_append (ifile, "SEQ cmd_alt_setup = " &
     // "alt_setup '=' option_list_expr")
call ifile_append (ifile, "KEY alt_setup")
call ifile_append (ifile, "ALT option_list_expr = " &
     // "grouped_option_list | option_list")
call ifile_append (ifile, "GRO grouped_option_list = ( option_list_expr )")
call ifile_append (ifile, "LIS option_list = options+")
call ifile_append (ifile, "SEQ cmd_open_out = open_out open_arg options?")
call ifile_append (ifile, "SEQ cmd_close_out = close_out open_arg options?")
call ifile_append (ifile, "KEY open_out")
call ifile_append (ifile, "KEY close_out")
```

```
call ifile_append (ifile, "ARG open_arg = (sexpr)")
call ifile_append (ifile, "SEQ cmd_printf = printf_cmd options?")
call ifile_append (ifile, "SEQ printf_cmd = printf_clause sprintf_args?")
call ifile_append (ifile, "SEQ printf_clause = printf sexpr")
call ifile_append (ifile, "KEY printf")
call ifile_append (ifile, "SEQ cmd_record = record_cmd")
call ifile_append (ifile, "SEQ cmd_unstable = " &
     // "unstable cexpr unstable_arg options?")
call ifile_append (ifile, "KEY unstable")
call ifile_append (ifile, "ARG unstable_arg = ( proc_id* )")
call ifile_append (ifile, "SEQ cmd_stable = stable stable_list options?")
call ifile_append (ifile, "KEY stable")
call ifile_append (ifile, "LIS stable_list = cexpr+")
call ifile_append (ifile, "KEY polarized")
call ifile_append (ifile, "SEQ cmd_polarized = polarized polarized_list options?")
call ifile_append (ifile, "LIS polarized_list = cexpr+")
call ifile_append (ifile, "KEY unpolarized")
call ifile_append (ifile, "SEQ cmd_unpolarized = unpolarized unpolarized_list options?")
call ifile_append (ifile, "LIS unpolarized_list = cexpr+")
call ifile_append (ifile, "SEQ cmd_simulate = " &
     // "simulate proc_arg options?")
call ifile_append (ifile, "KEY simulate")
call ifile_append (ifile, "SEQ cmd_rescan = " &
     // "rescan sexpr proc_arg options?")
call ifile_append (ifile, "KEY rescan")
call ifile_append (ifile, "SEQ cmd_scan = scan scan_var scan_body?")
call ifile_append (ifile, "KEY scan")
call ifile_append (ifile, "ALT scan_var = " &
     // "scan_log_decl | scan_log | " &
     // "scan_int | scan_real | scan_complex | scan_num | " &
     // "scan_string_decl | scan_string | scan_alias | " &
     // "scan_cuts | scan_weight | " &
     // "scan_scale | scan_ren_scale | scan_fac_scale | " &
     // "scan_selection | scan_reweight | scan_analysis | " &
     // "scan_model | scan_library")
call ifile_append (ifile, "SEQ scan_log_decl = logical scan_log")
call ifile_append (ifile, "SEQ scan_log = '?' var_name '=' scan_log_arg")
call ifile_append (ifile, "ARG scan_log_arg = ( lexpr* )")
call ifile_append (ifile, "SEQ scan_int = int var_name '=' scan_num_arg")
call ifile_append (ifile, "SEQ scan_real = real var_name '=' scan_num_arg")
call ifile_append (ifile, "SEQ scan_complex = " &
     // "complex var_name '=' scan_num_arg")
call ifile_append (ifile, "SEQ scan_num = var_name '=' scan_num_arg")
call ifile_append (ifile, "ARG scan_num_arg = ( range* )")
call ifile_append (ifile, "ALT range = grouped_range | range_expr")
call ifile_append (ifile, "GRO grouped_range = ( range_expr )")
call ifile_append (ifile, "SEQ range_expr = expr range_spec?")
call ifile_append (ifile, "SEQ range_spec = '=>' expr step_spec?")
call ifile_append (ifile, "SEQ step_spec = step_op expr")
call ifile_append (ifile, "ALT step_op = " &
     // "'/+' | '/-' | '/*' | '//' | '/+/' | '/*/'")
call ifile_append (ifile, "KEY '/+'")
call ifile_append (ifile, "KEY '/-'")
call ifile_append (ifile, "KEY '/*'")
```

```
    call ifile_append (ifile, "KEY '//'")
    call ifile_append (ifile, "KEY '/+/'")
    call ifile_append (ifile, "KEY '/*/'")
    call ifile_append (ifile, "SEQ scan_string_decl = string scan_string")
    call ifile_append (ifile, "SEQ scan_string = " &
         // "'$' var_name '=' scan_string_arg")
    call ifile_append (ifile, "ARG scan_string_arg = ( sexpr* )")
    call ifile_append (ifile, "SEQ scan_alias = " &
         // "alias var_name '=' scan_alias_arg")
    call ifile_append (ifile, "ARG scan_alias_arg = ( cexpr* )")
    call ifile_append (ifile, "SEQ scan_cuts = cuts '=' scan_lexpr_arg")
    call ifile_append (ifile, "ARG scan_lexpr_arg = ( lexpr* )")
    call ifile_append (ifile, "SEQ scan_scale = scale '=' scan_expr_arg")
    call ifile_append (ifile, "ARG scan_expr_arg = ( expr* )")
    call ifile_append (ifile, "SEQ scan_fac_scale = " &
         // "factorization_scale '=' scan_expr_arg")
    call ifile_append (ifile, "SEQ scan_ren_scale = " &
         // "renormalization_scale '=' scan_expr_arg")
    call ifile_append (ifile, "SEQ scan_weight = weight '=' scan_expr_arg")
    call ifile_append (ifile, "SEQ scan_selection = selection '=' scan_lexpr_arg")
    call ifile_append (ifile, "SEQ scan_reweight = reweight '=' scan_expr_arg")
    call ifile_append (ifile, "SEQ scan_analysis = analysis '=' scan_lexpr_arg")
    call ifile_append (ifile, "SEQ scan_model = model '=' scan_model_arg")
    call ifile_append (ifile, "ARG scan_model_arg = ( model_name* )")
    call ifile_append (ifile, "SEQ scan_library = library '=' scan_library_arg")
    call ifile_append (ifile, "ARG scan_library_arg = ( lib_name* )")
    call ifile_append (ifile, "GRO scan_body = '{' command_list '}'")
    call ifile_append (ifile, "SEQ cmd_if = " &
         // "if lexpr then command_list elsif_clauses else_clause endif")
    call ifile_append (ifile, "SEQ elsif_clauses = cmd_elsif*")
    call ifile_append (ifile, "SEQ cmd_elsif = elsif lexpr then command_list")
    call ifile_append (ifile, "SEQ else_clause = cmd_else?")
    call ifile_append (ifile, "SEQ cmd_else = else command_list")
    call ifile_append (ifile, "SEQ cmd_include = include include_arg")
    call ifile_append (ifile, "KEY include")
    call ifile_append (ifile, "ARG include_arg = ( string_literal )")
    call ifile_append (ifile, "SEQ cmd_quit = quit_cmd quit_arg?")
    call ifile_append (ifile, "ALT quit_cmd = quit | exit")
    call ifile_append (ifile, "KEY quit")
    call ifile_append (ifile, "KEY exit")
    call ifile_append (ifile, "ARG quit_arg = ( expr )")
    call ifile_append (ifile, "SEQ cmd_write_analysis = " &
         // "write_analysis_clause options?")
    call ifile_append (ifile, "SEQ cmd_compile_analysis = " &
         // "compile_analysis_clause options?")
    call ifile_append (ifile, "SEQ write_analysis_clause = " &
         // "write_analysis write_analysis_arg?")
    call ifile_append (ifile, "SEQ compile_analysis_clause = " &
         // "compile_analysis write_analysis_arg?")
    call ifile_append (ifile, "KEY write_analysis")
    call ifile_append (ifile, "KEY compile_analysis")
    call ifile_append (ifile, "ARG write_analysis_arg = ( analysis_tag* )")
    call define_expr_syntax (ifile, particles=.true., analysis=.true.)
  end subroutine define_cmd_list_syntax
```

3131

⟨*Commands: public*⟩+≡
```
    public :: lexer_init_cmd_list
```

⟨*Commands: procedures*⟩+≡
```
    subroutine lexer_init_cmd_list (lexer, parent_lexer)
      type(lexer_t), intent(out) :: lexer
      type(lexer_t), intent(in), optional, target :: parent_lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "()[]{},;:&%?$@", &
           special_class = [ "+-*/^", "<>=~ " ] , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_cmd_list), &
           parent = parent_lexer)
    end subroutine lexer_init_cmd_list
```

23.2.6 Unit Tests

⟨*Commands: public*⟩+≡
```
    public :: commands_test
```

⟨*Commands: tests*⟩≡
```
    subroutine commands_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
```
⟨*Commands: execute tests*⟩
```
    end subroutine commands_test
```

Prepare Sindarin code

This routine parses an internal file, prints the parse tree, and returns a parse
node to the root. We use the routine in the tests below.

⟨*Commands: tests*⟩+≡
```
    subroutine parse_ifile (ifile, pn_root, u)
      type(ifile_t), intent(in) :: ifile
      type(parse_node_t), pointer, intent(out) :: pn_root
      integer, intent(in), optional :: u
      type(stream_t), target :: stream
      type(lexer_t), target :: lexer
      type(parse_tree_t) :: parse_tree

      call lexer_init_cmd_list (lexer)
      call stream_init (stream, ifile)
      call lexer_assign_stream (lexer, stream)

      call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
      if (present (u))  call parse_tree_write (parse_tree, u)
      pn_root => parse_tree_get_root_ptr (parse_tree)
```

```
      call stream_final (stream)
      call lexer_final (lexer)
    end subroutine parse_ifile
```

Empty command list

Compile and execute an empty command list. Should do nothing but test the integrity of the workflow.

⟨*Commands: execute tests*⟩≡
```
    call test (commands_1, "commands_1", &
         "empty command list", &
         u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_1 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_1"
      write (u, "(A)")  "*   Purpose: compile and execute empty command list"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Parse empty file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"

      if (associated (pn_root)) then
         call command_list%compile (pn_root, global)
      end if

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"

      call command_list%execute (global)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)
```

```
      call command_list%final ()
      call syntax_cmd_list_final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_1"

    end subroutine commands_1
```

Read model

Execute a `model` assignment.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_2, "commands_2", &
           "model", &
           u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_2 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_2"
      write (u, "(A)")  "*   Purpose: set model"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "Test"')

      call ifile_write (ifile, u)

      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
```

3134

```
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)
      write (u, "(A)")  "global model = '" &
           //  char (global%model%get_name ()) // "'"

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_2"

    end subroutine commands_2
```

Declare Process

Read a model, then declare a process. The process library is allocated explicitly. For the process definition, We take the default (`omega`) method. Since we do not compile, `O'MEGA` is not actually called.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_3, "commands_3", &
           "process declaration", &
           u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_3 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: commands_3"
      write (u, "(A)")  "*   Purpose: define process"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
```

```
   call global%global_init ()
   call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
        .false., is_known = .true.)

   allocate (lib)
   call lib%init (var_str ("lib_cmd3"))
   call global%add_prclib (lib)

   write (u, "(A)")  "*  Input file"
   write (u, "(A)")

   call ifile_append (ifile, 'model = "Test"')
   call ifile_append (ifile, 'process t3 = s, s => s, s')

   call ifile_write (ifile, u)

   write (u, "(A)")
   write (u, "(A)")  "*  Parse file"
   write (u, "(A)")

   call parse_ifile (ifile, pn_root, u)

   write (u, "(A)")
   write (u, "(A)")  "* Compile command list"
   write (u, "(A)")

   call command_list%compile (pn_root, global)
   call command_list%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Execute command list"
   write (u, "(A)")

   call command_list%execute (global)

   call global%prclib_stack%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call ifile_final (ifile)

   call command_list%final ()
   call global%final ()
   call syntax_cmd_list_final ()
   call syntax_model_file_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: commands_3"

end subroutine commands_3
```

Compile Process

Read a model, then declare a process and compile the library. The process library is allocated explicitly. For the process definition, We take the default (`unit_test`) method. There is no external code, so compilation of the library is merely a formal status change.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_4, "commands_4", &
        "compilation", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_4 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: commands_4"
      write (u, "(A)")  "*   Purpose: define process and compile library"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()
      call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known=.true.)

      allocate (lib)
      call lib%init (var_str ("lib_cmd4"))
      call global%add_prclib (lib)

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "Test"')
      call ifile_append (ifile, 'process t4 = s, s => s, s')
      call ifile_append (ifile, 'compile ("lib_cmd4")')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")
```

3137

```
      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)

      call global%prclib_stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_4"

    end subroutine commands_4
```

Integrate Process

Read a model, then declare a process, compile the library, and integrate over phase space. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_5, "commands_5", &
         "integration", &
         u, results)
```
⟨*Commands: tests*⟩+≡
```
    subroutine commands_5 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: commands_5"
      write (u, "(A)")  "*   Purpose: define process, iterations, and integrate"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")
```

3138

```
call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()
call var_list_set_string (global%var_list, var_str ("$method"), &
     var_str ("unit_test"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("$phs_method"), &
     var_str ("single"), is_known=.true.)
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("midpoint"), is_known=.true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known=.true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
call var_list_set_real (global%var_list, var_str ("sqrts"), &
     1000._default, is_known=.true.)
call var_list_set_int (global%var_list, var_str ("seed"), &
     0, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd5"))
call global%add_prclib (lib)

write (u, "(A)")  "*  Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process t5 = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (t5)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call reset_interaction_counter ()
call command_list%execute (global)

call global%it_list%write (u)
```

```
      write (u, "(A)")
      call global%process_stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_5"

    end subroutine commands_5
```

Variables

Set intrinsic and user-defined variables.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_6, "commands_6", &
         "variables", &
         u, results)
```
⟨*Commands: tests*⟩+≡
```
    subroutine commands_6 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_6"
      write (u, "(A)")  "*   Purpose: define and set variables"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()
      call global%write_vars (u, [ &
           var_str ("$run_id"), &
           var_str ("?unweighted"), &
           var_str ("sqrts")])

      write (u, "(A)")
      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, '$run_id = "run1"')
```

3140

```
    call ifile_append (ifile, '?unweighted = false')
    call ifile_append (ifile, 'sqrts = 1000')
    call ifile_append (ifile, 'int j = 10')
    call ifile_append (ifile, 'real x = 1000.')
    call ifile_append (ifile, 'complex z = 5')
    call ifile_append (ifile, 'string $text = "abcd"')
    call ifile_append (ifile, 'logical ?flag = true')

    call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

    call command_list%execute (global)

    call global%write_vars (u, [ &
         var_str ("$run_id"), &
         var_str ("?unweighted"), &
         var_str ("sqrts"), &
         var_str ("j"), &
         var_str ("x"), &
         var_str ("z"), &
         var_str ("$text"), &
         var_str ("?flag")])


  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call syntax_cmd_list_final ()
    call global%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_6"

end subroutine commands_6
```

Process library

Open process libraries explicitly.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_7, "commands_7", &
        "process library", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_7 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_7"
      write (u, "(A)")  "*   Purpose: declare process libraries"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()
      call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
           .false., is_known = .true.)
      global%os_data%fc = "Fortran-compiler"
      global%os_data%fcflags = "Fortran-flags"

      write (u, "(A)")
      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'library = "lib_cmd7_1"')
      call ifile_append (ifile, 'library = "lib_cmd7_2"')
      call ifile_append (ifile, 'library = "lib_cmd7_1"')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
```

3142

```
        write (u, "(A)")

        call command_list%execute (global)

        call global%write_libraries (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call ifile_final (ifile)

        call command_list%final ()
        call syntax_cmd_list_final ()
        call global%final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: commands_7"

      end subroutine commands_7
```

Generate events

Read a model, then declare a process, compile the library, and generate weighted events. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_8, "commands_8", &
           "event generation", &
           u, results)
```

⟨*Commands: tests*⟩+≡
```
      subroutine commands_8 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root
        type(prclib_entry_t), pointer :: lib

        write (u, "(A)")  "* Test output: commands_8"
        write (u, "(A)")  "*   Purpose: define process, integrate, generate events"
        write (u, "(A)")

        write (u, "(A)")  "*  Initialization"
        write (u, "(A)")

        call syntax_cmd_list_init ()
        call syntax_model_file_init ()
        call global%global_init ()
        call var_list_set_string (global%var_list, var_str ("$method"), &
             var_str ("unit_test"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("$phs_method"), &
             var_str ("single"), is_known=.true.)
```

```
call var_list_set_string (global%var_list, var_str ("$integration_method"),&
     var_str ("midpoint"), is_known=.true.)
call var_list_set_log (global%var_list, var_str ("?vis_history"),&
     .false., is_known=.true.)
call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
     .false., is_known = .true.)
call var_list_set_real (global%var_list, var_str ("sqrts"), &
     1000._default, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd8"))
call global%add_prclib (lib)

write (u, "(A)")  "*  Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_8_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (commands_8_p)')
call ifile_append (ifile, '?unweighted = false')
call ifile_append (ifile, 'n_events = 3')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'simulate (commands_8_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"

call command_list%execute (global)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
```

```
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_8"

  end subroutine commands_8
```

Define cuts

Declare a cut expression.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_9, "commands_9", &
         "cuts", &
         u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_9 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(string_t), dimension(0) :: no_vars

      write (u, "(A)")  "* Test output: commands_9"
      write (u, "(A)")  "*   Purpose: define cuts"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'cuts = all Pt > 0 [particle]')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)
```

3145

```
      write (u, "(A)")
      write (u, "(A)")   "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)

      call global%write (u, vars = no_vars)

      write (u, "(A)")
      write (u, "(A)")   "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")   "* Test output end: commands_9"

    end subroutine commands_9
```

Beams

Define beam setup.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_10, "commands_10", &
         "beams", &
         u, results)
```
⟨*Commands: tests*⟩+≡
```
    subroutine commands_10 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")   "* Test output: commands_10"
      write (u, "(A)")   "*   Purpose: define beams"
      write (u, "(A)")

      write (u, "(A)")   "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()

      write (u, "(A)")   "*  Input file"
      write (u, "(A)")
```

```
      call ifile_append (ifile, 'model = QCD')
      call ifile_append (ifile, 'sqrts = 1000')
      call ifile_append (ifile, 'beams = p, p')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)

      call global%write_beams (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_10"

    end subroutine commands_10
```

Structure functions

Define beam setup with structure functions

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_11, "commands_11", &
        "structure functions", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_11 (u)
```

```
integer, intent(in) :: u
type(ifile_t) :: ifile
type(command_list_t), target :: command_list
type(rt_data_t), target :: global
type(parse_node_t), pointer :: pn_root

write (u, "(A)")  "* Test output: commands_11"
write (u, "(A)")  "*   Purpose: define beams with structure functions"
write (u, "(A)")

write (u, "(A)")  "*  Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()

write (u, "(A)")  "*  Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = QCD')
call ifile_append (ifile, 'sqrts = 1100')
call ifile_append (ifile, 'beams = p, p => lhapdf => pdf_builtin, isr')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%write_beams (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call global%final ()
```

```
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_11"

   end subroutine commands_11
```

Rescan events

Read a model, then declare a process, compile the library, and generate weighted events. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration. Then, rescan the generated event sample.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_12, "commands_12", &
           "event rescanning", &
           u, results)
```

⟨*Commands: tests*⟩+≡
```
      subroutine commands_12 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root
        type(prclib_entry_t), pointer :: lib

        write (u, "(A)")  "* Test output: commands_12"
        write (u, "(A)")  "*   Purpose: generate events and rescan"
        write (u, "(A)")

        write (u, "(A)")  "*  Initialization"
        write (u, "(A)")

        call syntax_cmd_list_init ()
        call syntax_model_file_init ()
        call global%global_init ()
        call var_list_set_string (global%var_list, var_str ("$method"), &
             var_str ("unit_test"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("$phs_method"), &
             var_str ("single"), is_known=.true.)
        call var_list_set_string (global%var_list, var_str ("$integration_method"),&
             var_str ("midpoint"), is_known=.true.)
        call var_list_set_log (global%var_list, var_str ("?vis_history"),&
             .false., is_known=.true.)
        call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
             .false., is_known = .true.)
        call var_list_set_real (global%var_list, var_str ("sqrts"), &
             1000._default, is_known=.true.)

        allocate (lib)
        call lib%init (var_str ("lib_cmd12"))
```

```
    call global%add_prclib (lib)

    write (u, "(A)")  "*  Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'model = "Test"')
    call ifile_append (ifile, 'process commands_12_p = s, s => s, s')
    call ifile_append (ifile, 'compile')
    call ifile_append (ifile, 'iterations = 1:1000')
    call ifile_append (ifile, 'integrate (commands_12_p)')
    call ifile_append (ifile, '?unweighted = false')
    call ifile_append (ifile, 'n_events = 3')
    call ifile_append (ifile, '?read_raw = false')
    call ifile_append (ifile, 'simulate (commands_12_p)')
    call ifile_append (ifile, '?write_raw = false')
    call ifile_append (ifile, 'rescan "commands_12_p" (commands_12_p)')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"

    call command_list%execute (global)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_12"

end subroutine commands_12
```

Event Files

Set output formats for event files.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_13, "commands_13", &
        "event output formats", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_13 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib
      logical :: exist

      write (u, "(A)")  "* Test output: commands_13"
      write (u, "(A)")  "*   Purpose: generate events and rescan"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()
      call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known=.true.)
      call var_list_set_string (global%var_list, var_str ("$phs_method"), &
          var_str ("single"), is_known=.true.)
      call var_list_set_string (global%var_list, var_str ("$integration_method"),&
          var_str ("midpoint"), is_known=.true.)
      call var_list_set_real (global%var_list, var_str ("sqrts"), &
          1000._default, is_known=.true.)
      call var_list_set_log (global%var_list, var_str ("?vis_history"),&
          .false., is_known=.true.)
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
          .false., is_known = .true.)

      allocate (lib)
      call lib%init (var_str ("lib_cmd13"))
      call global%add_prclib (lib)

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "Test"')
      call ifile_append (ifile, 'process commands_13_p = s, s => s, s')
      call ifile_append (ifile, 'compile')
      call ifile_append (ifile, 'iterations = 1:1000')
      call ifile_append (ifile, 'integrate (commands_13_p)')
      call ifile_append (ifile, '?unweighted = false')
      call ifile_append (ifile, 'n_events = 1')
```

3151

```
      call ifile_append (ifile, '?read_raw = false')
      call ifile_append (ifile, 'sample_format = weight_stream')
      call ifile_append (ifile, 'simulate (commands_13_p)')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"

      call command_list%execute (global)

      write (u, "(A)")
      write (u, "(A)")  "* Verify output files"
      write (u, "(A)")

      inquire (file = "commands_13_p.evx", exist = exist)
      if (exist)  write (u, "(1x,A)")  "raw"

      inquire (file = "commands_13_p.weights.dat", exist = exist)
      if (exist)  write (u, "(1x,A)")  "weight_stream"

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_13"

    end subroutine commands_13
```

Compile Empty Libraries

(This is a regression test:) Declare two empty libraries and compile them.

⟨*Commands: execute tests*⟩+≡

```
          call test (commands_14, "commands_14", &
               "empty libraries", &
               u, results)
```

⟨*Commands: tests*⟩+≡

```
     subroutine commands_14 (u)
       integer, intent(in) :: u
       type(ifile_t) :: ifile
       type(command_list_t), target :: command_list
       type(rt_data_t), target :: global
       type(parse_node_t), pointer :: pn_root

       write (u, "(A)")  "* Test output: commands_14"
       write (u, "(A)")  "*   Purpose: define and compile empty libraries"
       write (u, "(A)")

       write (u, "(A)")  "* Initialization"
       write (u, "(A)")

       call syntax_cmd_list_init ()
       call global%global_init ()

       write (u, "(A)")  "* Input file"
       write (u, "(A)")

       call ifile_append (ifile, 'library = "lib1"')
       call ifile_append (ifile, 'library = "lib2"')
       call ifile_append (ifile, 'compile ()')

       call ifile_write (ifile, u)

       write (u, "(A)")
       write (u, "(A)")  "* Parse file"
       write (u, "(A)")

       call parse_ifile (ifile, pn_root)

       write (u, "(A)")  "* Compile command list"
       write (u, "(A)")

       call command_list%compile (pn_root, global)

       write (u, "(A)")  "* Execute command list"
       write (u, "(A)")

       call command_list%execute (global)

       call global%prclib_stack%write (u)

       write (u, "(A)")
       write (u, "(A)")  "* Cleanup"

       call ifile_final (ifile)

       call command_list%final ()
```

3153

```
      call syntax_cmd_list_final ()
      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_14"

   end subroutine commands_14
```

Compile Process

Read a model, then declare a process and compile the library. The process library is allocated explicitly. For the process definition, We take the default (`unit_test`) method. There is no external code, so compilation of the library is merely a formal status change.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_15, "commands_15", &
          "compilation", &
          u, results)
```

⟨*Commands: tests*⟩+≡
```
   subroutine commands_15 (u)
     integer, intent(in) :: u
     type(ifile_t) :: ifile
     type(command_list_t), target :: command_list
     type(rt_data_t), target :: global
     type(parse_node_t), pointer :: pn_root
     type(prclib_entry_t), pointer :: lib

     write (u, "(A)")  "* Test output: commands_15"
     write (u, "(A)")  "*   Purpose: define process and compile library"
     write (u, "(A)")

     write (u, "(A)")  "* Initialization"
     write (u, "(A)")

     call syntax_cmd_list_init ()
     call syntax_model_file_init ()
     call global%global_init ()
     call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known=.true.)
     call var_list_set_string (global%var_list, var_str ("$phs_method"), &
          var_str ("single"), is_known=.true.)
     call var_list_set_string (global%var_list, var_str ("$integration_method"),&
          var_str ("midpoint"), is_known=.true.)
     call var_list_set_real (global%var_list, var_str ("sqrts"), &
          1000._default, is_known=.true.)
     call var_list_set_log (global%var_list, var_str ("?vis_history"),&
          .false., is_known=.true.)
     call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
          .false., is_known = .true.)

     allocate (lib)
     call lib%init (var_str ("lib_cmd15"))
```

```
    call global%add_prclib (lib)

    write (u, "(A)")  "* Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'model = "Test"')
    call ifile_append (ifile, 'process t15 = s, s => s, s')
    call ifile_append (ifile, 'iterations = 1:1000')
    call ifile_append (ifile, 'integrate (t15)')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "* Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root)

    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)

    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%prclib_stack%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_15"

  end subroutine commands_15
```

Observable

Declare an observable, fill it and display.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_16, "commands_16", &
        "observables", &
        u, results)
```

⟨Commands: tests⟩+≡

```
   subroutine commands_16 (u)
     integer, intent(in) :: u
     type(ifile_t) :: ifile
     type(command_list_t), target :: command_list
     type(rt_data_t), target :: global
     type(parse_node_t), pointer :: pn_root

     write (u, "(A)")  "* Test output: commands_16"
     write (u, "(A)")  "*   Purpose: declare an observable"
     write (u, "(A)")

     write (u, "(A)")  "* Initialization"
     write (u, "(A)")

     call syntax_cmd_list_init ()
     call global%global_init ()

     write (u, "(A)")  "* Input file"
     write (u, "(A)")

     call ifile_append (ifile, '$obs_label = "foo"')
     call ifile_append (ifile, '$obs_unit = "cm"')
     call ifile_append (ifile, '$title = "Observable foo"')
     call ifile_append (ifile, '$description = "This is observable foo"')
     call ifile_append (ifile, 'observable foo')

     call ifile_write (ifile, u)

     write (u, "(A)")
     write (u, "(A)")  "* Parse file"
     write (u, "(A)")

     call parse_ifile (ifile, pn_root)

     write (u, "(A)")  "* Compile command list"
     write (u, "(A)")

     call command_list%compile (pn_root, global)

     call command_list%write (u)

     write (u, "(A)")
     write (u, "(A)")  "* Execute command list"
     write (u, "(A)")

     call command_list%execute (global)

     write (u, "(A)")  "* Record two data items"
     write (u, "(A)")

     call analysis_record_data (var_str ("foo"), 1._default)
     call analysis_record_data (var_str ("foo"), 3._default)
```

3156

```
         write (u, "(A)")  "* Display analysis store"
         write (u, "(A)")

         call analysis_write (u, verbose=.true.)

         write (u, "(A)")
         write (u, "(A)")  "* Cleanup"

         call ifile_final (ifile)

         call analysis_final ()
         call command_list%final ()
         call global%final ()
         call syntax_cmd_list_final ()
         call syntax_model_file_final ()

         write (u, "(A)")
         write (u, "(A)")  "* Test output end: commands_16"

       end subroutine commands_16
```

Histogram

Declare a histogram, fill it and display.

⟨*Commands: execute tests*⟩+≡
```
       call test (commands_17, "commands_17", &
            "histograms", &
            u, results)
```

⟨*Commands: tests*⟩+≡
```
       subroutine commands_17 (u)
         integer, intent(in) :: u
         type(ifile_t) :: ifile
         type(command_list_t), target :: command_list
         type(rt_data_t), target :: global
         type(parse_node_t), pointer :: pn_root
         type(string_t), dimension(3) :: name
         integer :: i

         write (u, "(A)")  "* Test output: commands_17"
         write (u, "(A)")  "*   Purpose: declare histograms"
         write (u, "(A)")

         write (u, "(A)")  "* Initialization"
         write (u, "(A)")

         call syntax_cmd_list_init ()
         call global%global_init ()

         write (u, "(A)")  "* Input file"
         write (u, "(A)")

         call ifile_append (ifile, '$obs_label = "foo"')
```

3157

```
call ifile_append (ifile, '$obs_unit = "cm"')
call ifile_append (ifile, '$title = "Histogram foo"')
call ifile_append (ifile, '$description = "This is histogram foo"')
call ifile_append (ifile, 'histogram foo (0,5,1)')
call ifile_append (ifile, '$title = "Histogram bar"')
call ifile_append (ifile, '$description = "This is histogram bar"')
call ifile_append (ifile, 'n_bins = 2')
call ifile_append (ifile, 'histogram bar (0,5)')
call ifile_append (ifile, '$title = "Histogram gee"')
call ifile_append (ifile, '$description = "This is histogram gee"')
call ifile_append (ifile, '?normalize_bins = true')
call ifile_append (ifile, 'histogram gee (0,5)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Record two data items"
write (u, "(A)")

name(1) = "foo"
name(2) = "bar"
name(3) = "gee"

do i = 1, 3
   call analysis_record_data (name(i), 0.1_default, &
        weight = 0.25_default)
   call analysis_record_data (name(i), 3.1_default)
   call analysis_record_data (name(i), 4.1_default, &
        excess = 0.5_default)
   call analysis_record_data (name(i), 7.1_default)
end do

write (u, "(A)")  "* Display analysis store"
write (u, "(A)")

call analysis_write (u, verbose=.true.)
```

```
      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call analysis_final ()
      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_17"

    end subroutine commands_17
```

Plot

Declare a plot, fill it and display contents.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_18, "commands_18", &
        "plots", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_18 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_18"
      write (u, "(A)")  "*   Purpose: declare a plot"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, '$obs_label = "foo"')
      call ifile_append (ifile, '$obs_unit = "cm"')
      call ifile_append (ifile, '$title = "Plot foo"')
      call ifile_append (ifile, '$description = "This is plot foo"')
      call ifile_append (ifile, '$x_label = "x axis"')
      call ifile_append (ifile, '$y_label = "y axis"')
      call ifile_append (ifile, '?x_log = false')
```

3159

```
call ifile_append (ifile, '?y_log = true')
call ifile_append (ifile, 'x_min = -1')
call ifile_append (ifile, 'x_max = 1')
call ifile_append (ifile, 'y_min = 0.1')
call ifile_append (ifile, 'y_max = 1000')
call ifile_append (ifile, 'plot foo')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Record two data items"
write (u, "(A)")

call analysis_record_data (var_str ("foo"), 0._default, 20._default, &
     xerr = 0.25_default)
call analysis_record_data (var_str ("foo"), 0.5_default, 0.2_default, &
     yerr = 0.07_default)
call analysis_record_data (var_str ("foo"), 3._default, 2._default)

write (u, "(A)")  "* Display analysis store"
write (u, "(A)")

call analysis_write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call analysis_final ()
call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
```

```
        write (u, "(A)")  "* Test output end: commands_18"

    end subroutine commands_18
```

Graph

Combine two (empty) plots to a graph.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_19, "commands_19", &
        "graphs", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_19 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_19"
      write (u, "(A)")  "*   Purpose: combine two plots to a graph"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'plot a')
      call ifile_append (ifile, 'plot b')
      call ifile_append (ifile, '$title = "Graph foo"')
      call ifile_append (ifile, '$description = "This is graph foo"')
      call ifile_append (ifile, 'graph foo = a & b')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "* Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root)

      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)

      call command_list%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Display analysis object"
    write (u, "(A)")

    call analysis_write (var_str ("foo"), u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_19"

  end subroutine commands_19
```

Record Data

Record data in previously allocated analysis objects.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_20, "commands_20", &
        "record data", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_20 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_20"
      write (u, "(A)")  "*   Purpose: record data"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization: create observable, histogram, plot"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()
```

3162

```
  call analysis_init_observable (var_str ("o"))
  call analysis_init_histogram (var_str ("h"), 0._default, 1._default, 3, &
       normalize_bins = .false.)
  call analysis_init_plot (var_str ("p"))

  write (u, "(A)")  "* Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'record o (1.234)')
  call ifile_append (ifile, 'record h (0.5)')
  call ifile_append (ifile, 'record p (1, 2)')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "* Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root)

  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)

  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  write (u, "(A)")  "* Display analysis object"
  write (u, "(A)")

  call analysis_write (u, verbose = .true.)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call analysis_final ()
  call command_list%final ()
  call global%final ()
  call syntax_cmd_list_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_20"

end subroutine commands_20
```

Analysis

Declare an analysis expression and use it to fill an observable during event generation.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_21, "commands_21", &
        "analysis expression", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_21 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: commands_21"
      write (u, "(A)")  "*   Purpose: create and use analysis expression"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization: create observable"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()
      call var_list_set_string (global%var_list, var_str ("$method"), &
          var_str ("unit_test"), is_known=.true.)
      call var_list_set_string (global%var_list, var_str ("$phs_method"), &
          var_str ("single"), is_known=.true.)
      call var_list_set_string (global%var_list, var_str ("$integration_method"),&
          var_str ("midpoint"), is_known=.true.)
      call var_list_set_log (global%var_list, var_str ("?vis_history"),&
          .false., is_known=.true.)
      call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
          .false., is_known = .true.)
      call var_list_set_real (global%var_list, var_str ("sqrts"), &
          1000._default, is_known=.true.)

      allocate (lib)
      call lib%init (var_str ("lib_cmd8"))
      call global%add_prclib (lib)

      call analysis_init_observable (var_str ("m"))

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "Test"')
      call ifile_append (ifile, 'process commands_21_p = s, s => s, s')
```

```
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:100')
call ifile_append (ifile, 'integrate (commands_21_p)')
call ifile_append (ifile, '?unweighted = true')
call ifile_append (ifile, 'n_events = 3')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'observable m')
call ifile_append (ifile, 'analysis = record m (eval M [s])')
call ifile_append (ifile, 'simulate (commands_21_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Display analysis object"
write (u, "(A)")

call analysis_write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call analysis_final ()
call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_21"

end subroutine commands_21
```

Write Analysis

Write accumulated analysis data to file.

⟨*Commands: execute tests*⟩+≡

```
    call test (commands_22, "commands_22", &
        "write analysis", &
        u, results)
```

⟨*Commands: tests*⟩+≡

```
    subroutine commands_22 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: commands_22"
      write (u, "(A)")  "*   Purpose: write analysis data"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization: create observable"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      call analysis_init_observable (var_str ("m"))
      call analysis_record_data (var_str ("m"), 125._default)

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, '$out_file = "commands_22.dat"')
      call ifile_append (ifile, 'write_analysis')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "* Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root)

      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)

      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")
```

```
        call command_list%execute (global)

        write (u, "(A)")  "* Display analysis data"
        write (u, "(A)")

        u_file = free_unit ()
        open (u_file, file = "commands_22.dat", &
              action = "read", status = "old")
        do
           read (u_file, "(A)", iostat = iostat)  buffer
           if (iostat /= 0)  exit
           write (u, "(A)") trim (buffer)
        end do
        close (u_file)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call ifile_final (ifile)

        call analysis_final ()
        call command_list%final ()
        call global%final ()
        call syntax_cmd_list_final ()
        call syntax_model_file_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: commands_22"

      end subroutine commands_22
```

Compile Analysis

Write accumulated analysis data to file and compile.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_23, "commands_23", &
           "compile analysis", &
           u, results)
```
⟨*Commands: tests*⟩+≡
```
      subroutine commands_23 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root
        integer :: u_file, iostat
        character(256) :: buffer
        logical :: exist
        type(graph_options_t) :: graph_options

        write (u, "(A)")  "* Test output: commands_23"
```

```
write (u, "(A)")  "*   Purpose: write and compile analysis data"
write (u, "(A)")

write (u, "(A)")  "* Initialization: create and fill histogram"
write (u, "(A)")

call syntax_cmd_list_init ()
call global%global_init ()

call graph_options_init (graph_options)
call graph_options_set (graph_options, &
     title = var_str ("Histogram for test: commands 23"), &
     description = var_str ("This is a test."), &
     width_mm = 125, height_mm = 85)
call analysis_init_histogram (var_str ("h"), &
     0._default, 10._default, 2._default, .false., &
     graph_options = graph_options)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 3._default)
call analysis_record_data (var_str ("h"), 3._default)
call analysis_record_data (var_str ("h"), 3._default)
call analysis_record_data (var_str ("h"), 5._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, '$out_file = "commands_23.dat"')
call ifile_append (ifile, 'compile_analysis')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")
```

```
call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Delete Postscript output"
write (u, "(A)")

inquire (file = "commands_23.ps", exist = exist)
if (exist) then
   u_file = free_unit ()
   open (u_file, file = "commands_23.ps", action = "write", status = "old")
   close (u_file, status = "delete")
end if
inquire (file = "commands_23.ps", exist = exist)
write (u, "(1x,A,L1)")  "Postcript output exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* TeX file"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = "commands_23.tex", &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)
write (u, *)

inquire (file = "commands_23.ps", exist = exist)
write (u, "(1x,A,L1)")  "Postcript output exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call analysis_final ()
call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_23"
```

```
    end subroutine commands_23
```

Histogram

Declare a histogram, fill it and display.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_24, "commands_24", &
        "drawing options", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_24 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_24"
      write (u, "(A)")  "*   Purpose: check graph and drawing options"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, '$title = "Title"')
      call ifile_append (ifile, '$description = "Description"')
      call ifile_append (ifile, '$x_label = "X Label"')
      call ifile_append (ifile, '$y_label = "Y Label"')
      call ifile_append (ifile, 'graph_width_mm = 111')
      call ifile_append (ifile, 'graph_height_mm = 222')
      call ifile_append (ifile, 'x_min = -11')
      call ifile_append (ifile, 'x_max = 22')
      call ifile_append (ifile, 'y_min = -33')
      call ifile_append (ifile, 'y_max = 44')
      call ifile_append (ifile, '$gmlcode_bg = "GML Code BG"')
      call ifile_append (ifile, '$gmlcode_fg = "GML Code FG"')
      call ifile_append (ifile, '$fill_options = "Fill Options"')
      call ifile_append (ifile, '$draw_options = "Draw Options"')
      call ifile_append (ifile, '$err_options = "Error Options"')
      call ifile_append (ifile, '$symbol = "Symbol"')
      call ifile_append (ifile, 'histogram foo (0,1)')
      call ifile_append (ifile, 'plot bar')

      call ifile_write (ifile, u)

      write (u, "(A)")
```

```
      write (u, "(A)")  "* Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root)

      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)

      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)

      write (u, "(A)")  "* Display analysis store"
      write (u, "(A)")

      call analysis_write (u, verbose=.true.)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)

      call analysis_final ()
      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_24"

    end subroutine commands_24
```

Local Environment

Declare a local environment.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_25, "commands_25", &
        "local process environment", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_25 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
```

```
type(parse_node_t), pointer :: pn_root

write (u, "(A)")  "* Test output: commands_25"
write (u, "(A)")  "*   Purpose: declare local environment for process"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_cmd_list_init ()
call global%global_init ()
call var_list_set_log (global%var_list, var_str ("?omega_openmp"), &
     .false., is_known = .true.)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'library = "commands_25_lib"')
call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_25_p1 = g, g => g, g &
     &{ model = "QCD" }')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)
call global%write_libraries (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_25"
```

```
    end subroutine commands_25
```

Alternative Setups

Declare a list of alternative setups.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_26, "commands_26", &
         "alternative setups", &
         u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_26 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_26"
      write (u, "(A)")  "*   Purpose: declare alternative setups for simulation"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'int i = 0')
      call ifile_append (ifile, 'alt_setup = ({ i = 1 }, { i = 2 })')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "* Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root)

      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)

      call global%write_expr (u)
```

```
    write (u, "(A)")
    write (u, "(A)")   "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")   "* Test output end: commands_26"

  end subroutine commands_26
```

Unstable Particle

Define decay processes and declare a particle as unstable. Also check the commands stable, polarized, unpolarized.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_27, "commands_27", &
         "unstable and polarized particles", &
         u, results)
```

⟨*Commands: tests*⟩+≡
```
  subroutine commands_27 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")   "* Test output: commands_27"
    write (u, "(A)")   "*   Purpose: modify particle properties"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call syntax_model_file_init ()
    call global%global_init ()
    call var_list_set_string (global%var_list, var_str ("$method"), &
         var_str ("unit_test"), is_known=.true.)
    call var_list_set_string (global%var_list, var_str ("$phs_method"), &
         var_str ("single"), is_known=.true.)
    call var_list_set_string (global%var_list, var_str ("$integration_method"),&
         var_str ("midpoint"), is_known=.true.)
    call var_list_set_log (global%var_list, var_str ("?vis_history"),&
         .false., is_known=.true.)
    call var_list_set_log (global%var_list, var_str ("?integration_timer"),&
         .false., is_known = .true.)

    allocate (lib)
    call lib%init (var_str ("commands_27_lib"))
    call global%add_prclib (lib)
```

```
write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'ff = 0.4')
call ifile_append (ifile, 'process d1 = s => f, fbar')
call ifile_append (ifile, 'unstable s (d1)')
call ifile_append (ifile, 'polarized f, fbar')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Show model"
write (u, "(A)")

call model_write (global%model, u)

write (u, "(A)")
write (u, "(A)")  "* Extra Input"
write (u, "(A)")

call ifile_final (ifile)
call ifile_append (ifile, '?diagonal_decay = true')
call ifile_append (ifile, 'unstable s (d1)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")
```

```
call command_list%final ()
call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Show model"
write (u, "(A)")

call model_write (global%model, u)

write (u, "(A)")
write (u, "(A)")  "* Extra Input"
write (u, "(A)")

call ifile_final (ifile)
call ifile_append (ifile, '?isotropic_decay = true')
call ifile_append (ifile, 'unstable s (d1)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%final ()
call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Show model"
write (u, "(A)")

call model_write (global%model, u)

write (u, "(A)")
write (u, "(A)")  "* Extra Input"
write (u, "(A)")

call ifile_final (ifile)
```

```
        call ifile_append (ifile, 'stable s')
        call ifile_append (ifile, 'unpolarized f')

        call ifile_write (ifile, u)

        write (u, "(A)")
        write (u, "(A)")  "* Parse file"
        write (u, "(A)")

        call parse_ifile (ifile, pn_root)

        write (u, "(A)")  "* Compile command list"
        write (u, "(A)")

        call command_list%final ()
        call command_list%compile (pn_root, global)
        call command_list%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Execute command list"
        write (u, "(A)")

        call command_list%execute (global)

        write (u, "(A)")  "* Show model"
        write (u, "(A)")

        call model_write (global%model, u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call ifile_final (ifile)

        call command_list%final ()
        call global%final ()
        call syntax_model_file_init ()
        call syntax_cmd_list_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: commands_27"

    end subroutine commands_27
```

Quit the program

Quit the program.

⟨Commands: execute tests⟩+≡
```
    call test (commands_28, "commands_28", &
        "quit", &
        u, results)
```

⟨Commands: tests⟩+≡

```
subroutine commands_28 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root1, pn_root2
  type(string_t), dimension(0) :: no_vars

  write (u, "(A)")  "* Test output: commands_28"
  write (u, "(A)")  "*   Purpose: quit the program"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "*  Input file: quit without code"
  write (u, "(A)")

  call ifile_append (ifile, 'quit')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root1, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root1, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  call global%write (u, vars = no_vars)

  write (u, "(A)")
  write (u, "(A)")  "*  Input file: quit with code"
  write (u, "(A)")

  call ifile_final (ifile)
  call command_list%final ()
  call ifile_append (ifile, 'quit ( 3 + 4 )')
```

3178

```
        call ifile_write (ifile, u)

        write (u, "(A)")
        write (u, "(A)")  "*  Parse file"
        write (u, "(A)")

        call parse_ifile (ifile, pn_root2, u)

        write (u, "(A)")
        write (u, "(A)")  "* Compile command list"
        write (u, "(A)")

        call command_list%compile (pn_root2, global)
        call command_list%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Execute command list"
        write (u, "(A)")

        call command_list%execute (global)

        call global%write (u, vars = no_vars)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call ifile_final (ifile)

        call command_list%final ()
        call global%final ()
        call syntax_cmd_list_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: commands_28"

    end subroutine commands_28
```

SLHA interface

Testing commands steering the SLHA interface.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_29, "commands_29", &
        "SLHA interface", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_29 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
```

```
write (u, "(A)")  "* Test output: commands_29"
write (u, "(A)")  "*   Purpose: test SLHA interface"
write (u, "(A)")

write (u, "(A)")  "*  Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call syntax_slha_init ()
call global%global_init ()

write (u, "(A)")  "*  Model MSSM, read SLHA file"
write (u, "(A)")

call ifile_append (ifile, 'model = "MSSM"')
call ifile_append (ifile, '?slha_read_decays = true')
call ifile_append (ifile, 'read_slha ("sps1ap_decays.slha")')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Model MSSM, default values:"
write (u, "(A)")

call global%model%write (u, verbose = .false., &
     show_vertices = .false., show_particles = .false.)

write (u, "(A)")
write (u, "(A)")  "* Selected global variables"
write (u, "(A)")

call var_list_write_var (global%var_list, var_str ("mch1"), u)
call var_list_write_var (global%var_list, var_str ("wch1"), u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)
```

3180

```
        write (u, "(A)")  "* Model MSSM, values from SLHA file"
        write (u, "(A)")

        call global%model%write (u, verbose = .false., &
             show_vertices = .false., show_particles = .false.)

        write (u, "(A)")
        write (u, "(A)")  "* Selected global variables"
        write (u, "(A)")

        call var_list_write_var (global%var_list, var_str ("mch1"), u)
        call var_list_write_var (global%var_list, var_str ("wch1"), u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call ifile_final (ifile)

        call command_list%final ()
        call global%final ()
        call syntax_slha_final ()
        call syntax_model_file_final ()
        call syntax_cmd_list_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: commands_29"

      end subroutine commands_29
```

Expressions for scales

Declare a scale, factorization scale or factorization scale expression.

⟨*Commands: execute tests*⟩+≡
```
      call test (commands_30, "commands_30", &
           "scales", &
           u, results)
```
⟨*Commands: tests*⟩+≡
```
      subroutine commands_30 (u)
        integer, intent(in) :: u
        type(ifile_t) :: ifile
        type(command_list_t), target :: command_list
        type(rt_data_t), target :: global
        type(parse_node_t), pointer :: pn_root

        write (u, "(A)")  "* Test output: commands_30"
        write (u, "(A)")  "*   Purpose: define scales"
        write (u, "(A)")

        write (u, "(A)")  "*  Initialization"
        write (u, "(A)")

        call syntax_cmd_list_init ()
```

```
    call global%global_init ()

    write (u, "(A)")  "*  Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'scale = 200 GeV')
    call ifile_append (ifile, &
         'factorization_scale = eval Pt [particle]')
    call ifile_append (ifile, &
         'renormalization_scale = eval E [particle]')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%write_expr (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_30"

  end subroutine commands_30
```

Weight and reweight expressions

Declare an expression for event weights and reweighting.

⟨*Commands: execute tests*⟩+≡

```
      call test (commands_31, "commands_31", &
          "event weights/reweighting", &
          u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_31 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_31"
      write (u, "(A)")  "*   Purpose: define weight/reweight"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'weight = eval Pz [particle]')
      call ifile_append (ifile, 'reweight = eval M2 [particle]')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)

      call global%write_expr (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call ifile_final (ifile)
```

```
      call command_list%final ()
      call global%final ()
      call syntax_cmd_list_final ()
      call syntax_model_file_final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: commands_31"

    end subroutine commands_31
```

Selecting events

Declare an expression for selecting events in an analysis.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_32, "commands_32", &
        "event selection", &
        u, results)
```
⟨*Commands: tests*⟩+≡
```
    subroutine commands_32 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_32"
      write (u, "(A)")  "*   Purpose: define selection"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'selection = any PDG == 13 [particle]')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")
```

3184

```
        call command_list%compile (pn_root, global)
        call command_list%write (u)

        write (u, "(A)")
        write (u, "(A)")  "* Execute command list"
        write (u, "(A)")

        call command_list%execute (global)

        call global%write_expr (u)

        write (u, "(A)")
        write (u, "(A)")  "* Cleanup"

        call ifile_final (ifile)

        call command_list%final ()
        call global%final ()
        call syntax_cmd_list_final ()

        write (u, "(A)")
        write (u, "(A)")  "* Test output end: commands_32"

    end subroutine commands_32
```

Executing shell commands

Execute a shell command.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_33, "commands_33", &
        "execute shell command", &
        u, results)
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_33 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      integer :: u_file, iostat
      character(3) :: buffer

      write (u, "(A)")  "* Test output: commands_33"
      write (u, "(A)")  "*   Purpose: execute shell command"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()
```

```
   write (u, "(A)")  "*  Input file"
   write (u, "(A)")

   call ifile_append (ifile, 'exec ("echo foo >> bar")')

   call ifile_write (ifile, u)

   write (u, "(A)")
   write (u, "(A)")  "*  Parse file"
   write (u, "(A)")

   call parse_ifile (ifile, pn_root, u)

   write (u, "(A)")
   write (u, "(A)")  "* Compile command list"
   write (u, "(A)")

   call command_list%compile (pn_root, global)
   call command_list%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Execute command list"
   write (u, "(A)")

   call command_list%execute (global)
   u_file = free_unit ()
   open (u_file, file = "bar", &
        action = "read", status = "old")
   do
      read (u_file, "(A)", iostat = iostat)  buffer
      if (iostat /= 0) exit
   end do
   write (u, "(A,A)")  "should be 'foo': ", trim (buffer)
   close (u_file)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call ifile_final (ifile)

   call command_list%final ()
   call global%final ()
   call syntax_cmd_list_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: commands_33"

end subroutine commands_33
```

23.3 Toplevel module WHIZARD

⟨whizard.f90⟩≡
  ⟨*File header*⟩

  module whizard

  ⟨*Use file utils*⟩
  ⟨*Use strings*⟩
    use limits, only: VERSION_STRING !NODEP!
    use limits, only: EOF, BACKSLASH !NODEP!
    use diagnostics !NODEP!
    use unit_tests
    use ifiles
    use formats
    use md5
    use os_interface
    use cputime
    use lexers
    use parser
    use xml
    use colors
    use state_matrices
    use analysis
    use variables
    use user_code_interface
    use expressions
    use particles
    use models
    use sorting
    use evaluators
    use phs_forests
    use beams
    use polarizations
    use sm_qcd
    use sf_aux
    use sf_mappings
    use sf_base
    use sf_pdf_builtin
    use sf_lhapdf
    use sf_circe1
    use sf_circe2
    use sf_isr
    use sf_epa
    use sf_ewa
    use sf_escan
    use sf_beam_events
    use sf_user
    use phs_base
    use phs_single
    use phs_wood
    use rng_base
    use rng_tao
    use selectors

```
    use mci_base
    use mci_midpoint
    use mci_vamp
    use prclib_interfaces
    use particle_specifiers
    use process_libraries
    use prclib_stacks
    use hepmc_interface
    use jets
    use pdg_arrays
    use interactions
    use slha_interface
    use cascades
    use blha_driver
    use blha_config
    use prc_test
    use prc_template_me
    use prc_omega
    use subevt_expr
    use processes
    use process_stacks
    use event_transforms
    use decays
    use shower
    use events

    use eio_data
    use eio_base
    use eio_raw
    use eio_checkpoints
    use eio_lhef
    use eio_hepmc
    use eio_stdhep
    use eio_ascii
    use eio_weights

    use iterations
    use beam_structures
    use rt_data
    use dispatch
    use process_configurations
    use compilations
    use integrations
    use event_streams
    use simulations

    use commands
```

⟨*Standard module head*⟩

⟨*WHIZARD: public*⟩

⟨*WHIZARD: types*⟩

```
    save

  contains

  ⟨WHIZARD: procedures⟩

  end module whizard
```

23.3.1 Options

Here we introduce a wrapper that holds various user options, so they can transparently be passed from the main program to the `whizard` object. Most parameters are used for initializing the `global` state.

⟨*WHIZARD: public*⟩≡
```
    public :: whizard_options_t
```

⟨*WHIZARD: types*⟩≡
```
    type :: whizard_options_t
      type(string_t) :: preload_model
      type(string_t) :: default_lib
      type(string_t) :: preload_libraries
      logical :: rebuild_library = .false.
      logical :: recompile_library = .false.
      logical :: rebuild_user
      logical :: rebuild_phs = .false.
      logical :: rebuild_grids = .false.
      logical :: rebuild_events = .false.
    end type whizard_options_t
```

23.3.2 Parse tree stack

We collect all parse trees that we generate in the `whizard` object. To this end, we create a stack of parse trees. They must not be finalized before the `global` object is finalized, because items such as a cut definition may contain references to the parse tree from which they were generated.

⟨*WHIZARD: types*⟩+≡
```
    type, extends (parse_tree_t) :: pt_entry_t
      type(pt_entry_t), pointer :: previous => null ()
    end type pt_entry_t
```

This is the stack. Since we always prepend, we just need the `last` pointer.

⟨*WHIZARD: types*⟩+≡
```
    type :: pt_stack_t
      type(pt_entry_t), pointer :: last => null ()
     contains
     ⟨WHIZARD: pt stack: TBP⟩
    end type pt_stack_t
```

The finalizer is called at the very end.

⟨*WHIZARD: pt stack: TBP*⟩≡
```
    procedure :: final => pt_stack_final
```

⟨*WHIZARD: procedures*⟩≡
```
    subroutine pt_stack_final (pt_stack)
      class(pt_stack_t), intent(inout) :: pt_stack
      type(pt_entry_t), pointer :: current
      do while (associated (pt_stack%last))
         current => pt_stack%last
         pt_stack%last => current%previous
         call parse_tree_final (current%parse_tree_t)
         deallocate (current)
      end do
    end subroutine pt_stack_final
```

Create and push a new entry, keeping the previous ones.
⟨*WHIZARD: pt stack: TBP*⟩+≡
```
    procedure :: push => pt_stack_push
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine pt_stack_push (pt_stack, parse_tree)
      class(pt_stack_t), intent(inout) :: pt_stack
      type(parse_tree_t), intent(out), pointer :: parse_tree
      type(pt_entry_t), pointer :: current
      allocate (current)
      parse_tree => current%parse_tree_t
      current%previous => pt_stack%last
      pt_stack%last => current
    end subroutine pt_stack_push
```

23.3.3 The `whizard` object

An object of type `whizard_t` is the top-level wrapper for a WHIZARD instance. The object holds various default settings and the current state of the generator, the `global` object of type `rt_data_t`. This object contains, for instance, the list of variables and the process libraries.

Since components of the `global` subobject are frequently used as targets, the `whizard` object should also consistently carry the `target` attribute.

The various self-tests do no not use this object. They initialize only specific subsets of the system, according to their needs.

Note: we intend to allow several concurrent instances. In the current implementation, there are still a few obstacles to this: the model library and the syntax tables are global variables, and the error handling uses global state. This should be improved.

⟨*WHIZARD: public*⟩+≡
```
    public :: whizard_t
```

⟨*WHIZARD: types*⟩+≡
```
    type :: whizard_t
      type(whizard_options_t) :: options
      type(rt_data_t) :: global
      type(pt_stack_t) :: pt_stack
     contains
     ⟨WHIZARD: whizard: TBP⟩
    end type whizard_t
```

3190

23.3.4 Initialization and finalization

⟨*WHIZARD: whizard: TBP*⟩≡
```
    procedure :: init => whizard_init
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_init (whizard, options, paths, logfile)
      class(whizard_t), intent(out), target :: whizard
      type(whizard_options_t), intent(in) :: options
      type(paths_t), intent(in), optional :: paths
      type(string_t), intent(in), optional :: logfile
      call init_syntax_tables ()
      whizard%options = options
      call whizard%global%global_init (paths, logfile)
      call whizard%init_rebuild_flags ()
      call whizard%preload_model ()
      call whizard%preload_library ()
    end subroutine whizard_init
```

Apart from the global data which have been initialized above, the process and model lists need to be finalized.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: final => whizard_final
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_final (whizard)
      class(whizard_t), intent(inout), target :: whizard
      call whizard%global%final ()
      call whizard%pt_stack%final ()
  !!! JRR: WK please check (#529)
      !    call user_code_final ()
      call final_syntax_tables ()
    end subroutine whizard_final
```

Set the rebuild flags. They can be specified on the command line and set the initial value for the associated logical variables.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: init_rebuild_flags => whizard_init_rebuild_flags
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_init_rebuild_flags (whizard)
      class(whizard_t), intent(inout), target :: whizard
      associate (var_list => whizard%global%var_list, options => whizard%options)
        call var_list_append_log &
            (var_list, var_str ("?rebuild_library"), options%rebuild_library, &
            intrinsic=.true.)
        call var_list_append_log &
            (var_list, var_str ("?recompile_library"), &
            options%recompile_library, &
            intrinsic=.true.)
        call var_list_append_log &
            (var_list, var_str ("?rebuild_phase_space"), options%rebuild_phs, &
```

```
            intrinsic=.true.)
        call var_list_append_log &
             (var_list, var_str ("?rebuild_grids"), options%rebuild_grids, &
             intrinsic=.true.)
        call var_list_append_log &
             (var_list, var_str ("?rebuild_events"), options%rebuild_events, &
             intrinsic=.true.)
      end associate
    end subroutine whizard_init_rebuild_flags
```

This procedure preloads a model, if a model name is given.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: preload_model => whizard_preload_model
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_preload_model (whizard)
      class(whizard_t), intent(inout), target :: whizard
      type(string_t) :: model_name, filename
      model_name = whizard%options%preload_model
      if (model_name /= "") then
         filename = model_name // ".mdl"
         call whizard%global%read_model (model_name, filename)
         if (associated (whizard%global%model)) then
            call msg_message ("Preloaded model: " &
                 // char (model_name))
         else
            call msg_fatal ("Preloading model " // char (model_name) &
                 // " failed")
         end if
      else
         call msg_message ("No model preloaded")
      end if
    end subroutine whizard_preload_model
```

This procedure preloads a library, if a library name is given.

Note: This version just opens a new library with that name. It does not load (yet) an existing library on file, as previous WHIZARD versions would do.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: preload_library => whizard_preload_library
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_preload_library (whizard)
      class(whizard_t), intent(inout), target :: whizard
      type(string_t) :: library_name, libs
      type(string_t), dimension(:), allocatable :: libname_static
      type(prclib_entry_t), pointer :: lib_entry
      integer :: i
      call get_prclib_static (libname_static)
      do i = 1, size (libname_static)
         allocate (lib_entry)
         call lib_entry%init_static (libname_static(i))
         call whizard%global%add_prclib (lib_entry)
      end do
      libs = adjustl (whizard%options%preload_libraries)
```

```
      if (libs == "" .and. whizard%options%default_lib /= "") then
            allocate (lib_entry)
            call lib_entry%init (whizard%options%default_lib)
            call whizard%global%add_prclib (lib_entry)
            call msg_message ("Preloaded library: " // &
                 char (whizard%options%default_lib))
         end if
   SCAN_LIBS: do while (libs /= "")
      call split (libs, library_name, " ")
      if (library_name /= "") then
         allocate (lib_entry)
         call lib_entry%init (library_name)
         call whizard%global%add_prclib (lib_entry)
         call msg_message ("Preloaded library: " // char (library_name))
      end if
   end do SCAN_LIBS
 end subroutine whizard_preload_library
```

23.3.5 Initialization and finalization (old version)

These procedures initialize and finalize global variables. Most of them are collected in the `global` data record located here, the others are syntax tables located in various modules, which do not change during program execution. Furthermore, there is a global model list and a global process store, which get filled during program execution but are finalized here.

During initialization, we can preload a default model and initialize a default library for setting up processes. The default library is loaded if requested by the setup. Further libraries can be loaded as specified by command-line flags. Initialize/finalize the syntax tables used by WHIZARD:

⟨*WHIZARD: public*⟩+≡
```
    public :: init_syntax_tables
    public :: final_syntax_tables
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine init_syntax_tables ()
      call syntax_model_file_init ()
      call syntax_phs_forest_init ()
      call syntax_pexpr_init ()
      call syntax_slha_init ()
      call syntax_blha_contract_init ()
      call syntax_cmd_list_init ()
    end subroutine init_syntax_tables

    subroutine final_syntax_tables ()
      call syntax_model_file_final ()
      call syntax_phs_forest_final ()
      call syntax_pexpr_final ()
      call syntax_slha_final ()
      call syntax_blha_contract_final ()
      call syntax_cmd_list_final ()
    end subroutine final_syntax_tables
```

Write the syntax tables to external files.

⟨*WHIZARD: public*⟩+≡
```
    public :: write_syntax_tables
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine write_syntax_tables ()
      integer :: unit
      character(*), parameter :: file_model = "whizard.model_file.syntax"
      character(*), parameter :: file_phs = "whizard.phase_space_file.syntax"
      character(*), parameter :: file_pexpr = "whizard.prt_expressions.syntax"
      character(*), parameter :: file_slha = "whizard.slha.syntax"
      character(*), parameter :: file_sindarin = "whizard.sindarin.syntax"
      unit = free_unit ()
      print *, "Writing file '" // file_model // "'"
      open (unit=unit, file=file_model, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_model
      call syntax_model_file_write (unit)
      close (unit)
      print *, "Writing file '" // file_phs // "'"
      open (unit=unit, file=file_phs, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_phs
      call syntax_phs_forest_write (unit)
      close (unit)
      print *, "Writing file '" // file_pexpr // "'"
      open (unit=unit, file=file_pexpr, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_pexpr
      call syntax_pexpr_write (unit)
      close (unit)
      print *, "Writing file '" // file_slha // "'"
      open (unit=unit, file=file_slha, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_slha
      call syntax_slha_write (unit)
      close (unit)
      print *, "Writing file '" // file_sindarin // "'"
      open (unit=unit, file=file_sindarin, status="replace", action="write")
      write (unit, "(A)")  VERSION_STRING
      write (unit, "(A)")  "Syntax definition file: " // file_sindarin
      call syntax_cmd_list_write (unit)
      close (unit)
    end subroutine write_syntax_tables
```

23.3.6   Execute command lists

Process commands given on the command line, stored as an `ifile`. The whole input is read, compiled and executed as a whole.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: process_ifile => whizard_process_ifile
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_ifile (whizard, ifile, quit, quit_code)
      class(whizard_t), intent(inout), target :: whizard
      type(ifile_t), intent(in) :: ifile
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      call msg_message ("Reading commands given on the command line")
      call lexer_init_cmd_list (lexer)
      call stream_init (stream, ifile)
      call whizard%process_stream (stream, lexer, quit, quit_code)
      call stream_final (stream)
      call lexer_final (lexer)
    end subroutine whizard_process_ifile
```

Process standard input as a command list. The whole input is read, compiled
and executed as a whole.
⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: process_stdin => whizard_process_stdin
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_stdin (whizard, quit, quit_code)
      class(whizard_t), intent(inout), target :: whizard
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      call msg_message ("Reading commands from standard input")
      call lexer_init_cmd_list (lexer)
      call stream_init (stream, 5)
      call whizard%process_stream (stream, lexer, quit, quit_code)
      call stream_final (stream)
      call lexer_final (lexer)
    end subroutine whizard_process_stdin
```

Process a file as a command list.
⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: process_file => whizard_process_file
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_file (whizard, file, quit, quit_code)
      class(whizard_t), intent(inout), target :: whizard
      type(string_t), intent(in) :: file
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      logical :: exist
      call msg_message ("Reading commands from file '" // char (file) // "'")
      inquire (file=char(file), exist=exist)
      if (exist) then
         call lexer_init_cmd_list (lexer)
         call stream_init (stream, char (file))
```

```
      call whizard%process_stream (stream, lexer, quit, quit_code)
      call stream_final (stream)
      call lexer_final (lexer)
    else
      call msg_error ("File '" // char (file) // "' not found")
    end if
  end subroutine whizard_process_file
```

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: process_stream => whizard_process_stream
```
⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_process_stream (whizard, stream, lexer, quit, quit_code)
      class(whizard_t), intent(inout), target :: whizard
      type(stream_t), intent(inout), target :: stream
      type(lexer_t), intent(inout), target :: lexer
      logical, intent(out) :: quit
      integer, intent(out) :: quit_code
      type(parse_tree_t), pointer :: parse_tree
      type(command_list_t), target :: command_list
      call lexer_assign_stream (lexer, stream)
      call whizard%pt_stack%push (parse_tree)
      call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
      if (associated (parse_tree_get_root_ptr (parse_tree))) then
         whizard%global%lexer => lexer
         call command_list%compile (parse_tree_get_root_ptr (parse_tree), &
             whizard%global)
      end if
      call command_list%execute (whizard%global)
      call command_list%final ()
      quit = whizard%global%quit
      quit_code = whizard%global%quit_code
    end subroutine whizard_process_stream
```

23.3.7   The WHIZARD shell

This procedure implements interactive mode. One line is processed at a time.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: shell => whizard_shell
```
⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_shell (whizard, quit_code)
      class(whizard_t), intent(inout), target :: whizard
      integer, intent(out) :: quit_code
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      type(string_t) :: prompt1
      type(string_t) :: prompt2
      type(string_t) :: input
      type(string_t) :: extra
      integer :: last
      integer :: iostat
      logical :: mask_tmp
```

```
         logical :: quit
         call msg_message ("Launching interactive shell")
         call lexer_init_cmd_list (lexer)
         prompt1 = "whish? "
         prompt2 = "      > "
         COMMAND_LOOP: do
            call put (6, prompt1)
            call get (5, input, iostat=iostat)
            if (iostat > 0 .or. iostat == EOF) exit COMMAND_LOOP
            CONTINUE_INPUT: do
               last = len_trim (input)
               if (extract (input, last, last) /= BACKSLASH)  exit CONTINUE_INPUT
               call put (6, prompt2)
               call get (5, extra, iostat=iostat)
               if (iostat > 0) exit COMMAND_LOOP
               input = replace (input, last, extra)
            end do CONTINUE_INPUT
            call stream_init (stream, input)
            mask_tmp = mask_fatal_errors
            mask_fatal_errors = .true.
            call whizard%process_stream (stream, lexer, quit, quit_code)
            msg_count = 0
            mask_fatal_errors = mask_tmp
            call stream_final (stream)
            if (quit)  exit COMMAND_LOOP
         end do COMMAND_LOOP
         print *
         call lexer_final (lexer)
      end subroutine whizard_shell
```

23.3.8   Self-tests

This is for developers only, but needs a well-defined interface.

⟨*WHIZARD: public*⟩+≡
```
      public :: whizard_check
```

⟨*WHIZARD: procedures*⟩+≡
```
      subroutine whizard_check (check, results)
        type(string_t), intent(in) :: check
        type(test_results_t), intent(inout) :: results
        type(os_data_t) :: os_data
        integer :: u
        call os_data_init (os_data)
        u = free_unit ()
        open (u, file="whizard_check." // char (check) // ".log", &
             action="write", status="replace")
        call msg_message (repeat ('=', 76), 0)
        call msg_message ("Running self-test: " // char (check), 0)
        call msg_message (repeat ('-', 76), 0)
        select case (char (check))
        case ("analysis")
           call analysis_test (u, results)
        case ("beams")
```

```
         call beam_test (u, results)
case ("cascades")
         call cascade_test (u, results)
case ("colors")
         call color_test (u, results)
case ("evaluators")
       call evaluator_test (u, results)
case ("expressions")
         call expressions_test (u, results)
case ("formats")
         call format_test (u, results)
case ("hepmc")
         call hepmc_test (u, results)
case ("jets")
         call jets_test (u, results)
case ("pdg_arrays")
         call pdg_arrays_test (u, results)
case ("interactions")
         call interaction_test (u, results)
case ("lexers")
         call lexer_test (u, results)
case ("os_interface")
         call os_interface_test (u, results)
case ("cputime")
         call cputime_test (u, results)
case ("parser")
         call parse_test (u, results)
case ("sorting")
         call sorting_test (u, results)
case ("md5")
         call md5_test (u, results)
case ("xml")
         call xml_test (u, results)
case ("sm_qcd")
         call sm_qcd_test (u, results)
case ("models")
         call models_test (u, results)
case ("particles")
         call particles_test (u, results)
case ("polarizations")
         call polarization_test (u, results)
case ("sf_aux")
         call sf_aux_test (u, results)
case ("sf_mappings")
         call sf_mappings_test (u, results)
case ("sf_base")
         call sf_base_test (u, results)
case ("sf_pdf_builtin")
         call sf_pdf_builtin_test (u, results)
case ("sf_lhapdf")
         call sf_lhapdf_test (u, results)
case ("sf_isr")
         call sf_isr_test (u, results)
case ("sf_epa")
```

```
      call sf_epa_test (u, results)
case ("sf_ewa")
      call sf_ewa_test (u, results)
case ("sf_circe1")
      call sf_circe1_test (u, results)
case ("sf_circe2")
      call sf_circe2_test (u, results)
case ("sf_beam_events")
      call sf_beam_events_test (u, results)
case ("sf_escan")
      call sf_escan_test (u, results)
case ("phs_base")
      call phs_base_test (u, results)
case ("phs_single")
      call phs_single_test (u, results)
case ("phs_forests")
      call phs_forest_test (u, results)
case ("phs_wood")
      call phs_wood_test (u, results)
case ("phs_wood_vis")
      call phs_wood_vis_test (u, results)
case ("mci_base")
      call mci_base_test (u, results)
case ("rng_base")
      call rng_base_test (u, results)
case ("rng_tao")
      call rng_tao_test (u, results)
case ("selectors")
      call selectors_test (u, results)
case ("mci_midpoint")
      call mci_midpoint_test (u, results)
case ("mci_vamp")
      call mci_vamp_test (u, results)
case ("prclib_interfaces")
      call prclib_interfaces_test (u, results)
case ("particle_specifiers")
      call particle_specifiers_test (u, results)
case ("process_libraries")
      call process_libraries_test (u, results)
case ("prclib_stacks")
      call prclib_stacks_test (u, results)
case ("slha_interface")
      call slha_test (u, results)
case ("state_matrices")
      call state_matrix_test (u, results)
case ("prc_test")
      call prc_test_test (u, results)
case ("subevt_expr")
      call subevt_expr_test (u, results)
case ("processes")
      call processes_test (u, results)
case ("process_stacks")
      call process_stacks_test (u, results)
case ("event_transforms")
```

```
         call event_transforms_test (u, results)
case ("decays")
         call decays_test (u, results)
case ("shower")
         call shower_test (u, results)
case ("events")
         call events_test (u, results)
case ("prc_template_me")
         call prc_template_me_test (u, results)
case ("prc_omega")
         call prc_omega_test (u, results)
case ("prc_omega_diags")
         call prc_omega_diags_test (u, results)
case ("eio_data")
         call eio_data_test (u, results)
case ("eio_base")
         call eio_base_test (u, results)
case ("eio_raw")
         call eio_raw_test (u, results)
case ("eio_checkpoints")
         call eio_checkpoints_test (u, results)
case ("eio_lhef")
         call eio_lhef_test (u, results)
case ("eio_hepmc")
         call eio_hepmc_test (u, results)
case ("eio_stdhep")
         call eio_stdhep_test (u, results)
case ("eio_ascii")
         call eio_ascii_test (u, results)
case ("eio_weights")
         call eio_weights_test (u, results)
case ("iterations")
         call iterations_test (u, results)
case ("beam_structures")
         call beam_structures_test (u, results)
case ("rt_data")
         call rt_data_test (u, results)
case ("dispatch")
         call dispatch_test (u, results)
case ("process_configurations")
         call process_configurations_test (u, results)
case ("compilations")
         call compilations_test (u, results)
case ("compilations_static")
         call compilations_static_test (u, results)
case ("integrations")
         call integrations_test (u, results)
case ("integrations_history")
         call integrations_history_test (u, results)
case ("event_streams")
         call event_streams_test (u, results)
case ("simulations")
         call simulations_test (u, results)
case ("commands")
```

```
                      call commands_test (u, results)
               case ("all")
                      call analysis_test (u, results)
                      call beam_test (u, results)
                      call md5_test (u, results)
                      call lexer_test (u, results)
                      call sorting_test (u, results)
                      call parse_test (u, results)
                      call color_test (u, results)
                      call evaluator_test (u, results)
                      call expressions_test (u, results)
                      call format_test (u, results)
                      call hepmc_test (u, results)
                      call jets_test (u, results)
                      call os_interface_test (u, results)
                      call cputime_test (u, results)
                      call interaction_test (u, results)
                      call xml_test (u, results)
                      call sm_qcd_test (u, results)
                      call models_test (u, results)
                      call particles_test (u, results)
                      call polarization_test (u, results)
                      call sf_aux_test (u, results)
                      call sf_mappings_test (u, results)
                      call sf_base_test (u, results)
                      call sf_pdf_builtin_test (u, results)
                      call sf_lhapdf_test (u, results)
                      call sf_isr_test (u, results)
                      call sf_epa_test (u, results)
                      call sf_ewa_test (u, results)
                      call sf_circe1_test (u, results)
                      call sf_circe2_test (u, results)
                      call sf_beam_events_test (u, results)
                      call sf_escan_test (u, results)
                      call phs_base_test (u, results)
                      call phs_single_test (u, results)
                      call phs_forest_test (u, results)
                      call phs_wood_test (u, results)
                      call phs_wood_vis_test (u, results)
                      call rng_base_test (u, results)
                      call cascade_test (u, results)
                      call rng_tao_test (u, results)
                      call selectors_test (u, results)
                      call mci_base_test (u, results)
                      call mci_midpoint_test (u, results)
                      call mci_vamp_test (u, results)
                      call prclib_interfaces_test (u, results)
                      call particle_specifiers_test (u, results)
                      call process_libraries_test (u, results)
                      call prclib_stacks_test (u, results)
                      call slha_test (u, results)
                      call state_matrix_test (u, results)
                      call prc_test_test (u, results)
                      call subevt_expr_test (u, results)
```

```
      call processes_test (u, results)
      call process_stacks_test (u, results)
      call event_transforms_test (u, results)
      call decays_test (u, results)
      call shower_test (u, results)
      call events_test (u, results)
      call prc_omega_test (u, results)
      call prc_omega_diags_test (u, results)
      call prc_template_me_test (u, results)
      call eio_data_test (u, results)
      call eio_base_test (u, results)
      call eio_raw_test (u, results)
      call eio_checkpoints_test (u, results)
      call eio_lhef_test (u, results)
      call eio_hepmc_test (u, results)
      call eio_stdhep_test (u, results)
      call eio_ascii_test (u, results)
      call eio_weights_test (u, results)
      call iterations_test (u, results)
      call beam_structures_test (u, results)
      call rt_data_test (u, results)
      call dispatch_test (u, results)
      call process_configurations_test (u, results)
      call compilations_test (u, results)
      call compilations_static_test (u, results)
      call integrations_test (u, results)
      call integrations_history_test (u, results)
      call event_streams_test (u, results)
      call simulations_test (u, results)
      call commands_test (u, results)
   case default
      call msg_fatal ("Self-test '" // char (check) // "' not implemented.")
   end select
   close (u)
end subroutine whizard_check
```

23.4   Driver program

The main program handles command options, initializes the environment, and runs WHIZARD in a particular mode (interactive, file, standard input).

⟨*Limits: public parameters*⟩+≡
```
    integer, parameter, public :: CMDLINE_ARG_LEN = 1000
```

⟨`main.f90`⟩≡
  ⟨*File header*⟩

  program main

  ⟨*Use strings*⟩
    use system_dependencies !NODEP!
    use limits, only: CMDLINE_ARG_LEN !NODEP!
    use diagnostics !NODEP!

```
use unit_tests
use ifiles
use os_interface
use whizard

implicit none

! Main program variable declarations
character(CMDLINE_ARG_LEN) :: arg
character(2) :: option
type(string_t) :: long_option, value
integer :: i, j, arg_len, arg_status
logical :: look_for_options
logical :: interactive
logical :: banner
type(string_t) :: files, this, model, default_lib, library, libraries
type(string_t) :: check, checks, logfile
type(test_results_t) :: test_results
logical :: success
logical :: user_code_enable = .false.
integer :: n_user_src = 0, n_user_lib = 0
type(string_t) :: user_src, user_lib, user_target
type(paths_t) :: paths
logical :: rebuild_library, rebuild_user
logical :: rebuild_phs, rebuild_grids, rebuild_events
logical :: recompile_library
type(ifile_t) :: commands
type(string_t) :: command

type(whizard_options_t), allocatable :: options
type(whizard_t), allocatable, target :: whizard_instance

! Exit status
logical :: quit = .false.
integer :: quit_code = 0

! Initial values
look_for_options = .true.
interactive = .false.
files = ""
model = "SM"
default_lib = "default_lib"
library = ""
libraries = ""
banner = .true.
logging = .true.
logfile = "whizard.log"
check = ""
checks = ""
user_src = ""
user_lib = ""
user_target = ""
rebuild_library = .false.
rebuild_user = .false.
```

```fortran
rebuild_phs = .false.
rebuild_grids = .false.
rebuild_events = .false.
recompile_library = .false.
call paths_init (paths)

! Read and process options
i = 0
SCAN_CMDLINE: do
   i = i + 1
   call get_command_argument (i, arg, arg_len, arg_status)
   select case (arg_status)
   case (0)
   case (-1)
      call msg_error (" Command argument truncated: '" // arg // "'")
   case default
      exit SCAN_CMDLINE
   end select
   if (look_for_options) then
      select case (arg(1:2))
      case ("--")
         value = trim (arg)
         call split (value, long_option, "=")
         select case (char (long_option))
         case ("--version")
            call no_option_value (long_option, value)
            call print_version (); stop
         case ("--help")
            call no_option_value (long_option, value)
            call print_usage (); stop
         case ("--prefix")
            paths%prefix = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--exec-prefix")
            paths%exec_prefix = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--bindir")
            paths%bindir = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--libdir")
            paths%libdir = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--includedir")
            paths%includedir = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--datarootdir")
            paths%datarootdir = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--libtool")
            paths%libtool = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
         case ("--lhapdfdir")
            paths%lhapdfdir = get_option_value (i, long_option, value)
            cycle SCAN_CMDLINE
```

```
case ("--check")
   check = get_option_value (i, long_option, value)
   checks = checks // " " // check
   cycle SCAN_CMDLINE
case ("--execute")
   command = get_option_value (i, long_option, value)
   call ifile_append (commands, command)
   cycle SCAN_CMDLINE
case ("--interactive")
   call no_option_value (long_option, value)
   interactive = .true.
   cycle SCAN_CMDLINE
case ("--library")
   library = get_option_value (i, long_option, value)
   libraries = libraries // " " // library
   cycle SCAN_CMDLINE
case ("--no-library")
   call no_option_value (long_option, value)
   default_lib = ""
   library = ""
   libraries = ""
   cycle SCAN_CMDLINE
case ("--localprefix")
   paths%localprefix = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--logfile")
   logfile = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--no-logfile")
   call no_option_value (long_option, value)
   logfile = ""
   cycle SCAN_CMDLINE
case ("--logging")
   call no_option_value (long_option, value)
   logging = .true.
   cycle SCAN_CMDLINE
case ("--no-logging")
   call no_option_value (long_option, value)
   logging = .false.
   cycle SCAN_CMDLINE
case ("--banner")
   call no_option_value (long_option, value)
   banner = .true.
   cycle SCAN_CMDLINE
case ("--no-banner")
   call no_option_value (long_option, value)
   banner = .false.
   cycle SCAN_CMDLINE
case ("--model")
   model = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--no-model")
   call no_option_value (long_option, value)
   model = ""
```

```
         cycle SCAN_CMDLINE
case ("--rebuild")
   call no_option_value (long_option, value)
   rebuild_library = .true.
   rebuild_user = .true.
   rebuild_phs = .true.
   rebuild_grids = .true.
   rebuild_events = .true.
   cycle SCAN_CMDLINE
case ("--no-rebuild")
   call no_option_value (long_option, value)
   rebuild_library = .false.
   recompile_library = .false.
   rebuild_user = .false.
   rebuild_phs = .false.
   rebuild_grids = .false.
   rebuild_events = .false.
   cycle SCAN_CMDLINE
case ("--rebuild-library")
   call no_option_value (long_option, value)
   rebuild_library = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-user")
   call no_option_value (long_option, value)
   rebuild_user = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-phase-space")
   call no_option_value (long_option, value)
   rebuild_phs = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-grids")
   call no_option_value (long_option, value)
   rebuild_grids = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-events")
   call no_option_value (long_option, value)
   rebuild_events = .true.
   cycle SCAN_CMDLINE
case ("--recompile")
   call no_option_value (long_option, value)
   recompile_library = .true.
   rebuild_grids = .true.
   cycle SCAN_CMDLINE
case ("--user")
   user_code_enable = .true.
   cycle SCAN_CMDLINE
case ("--user-src")
   if (user_src == "") then
      user_src = get_option_value (i, long_option, value)
   else
      user_src = user_src // " " &
            // get_option_value (i, long_option, value)
   end if
   n_user_src = n_user_src + 1
```

3206

```
            cycle SCAN_CMDLINE
      case ("--user-lib")
         if (user_lib == "") then
            user_lib = get_option_value (i, long_option, value)
         else
            user_lib = user_lib // " " &
                  // get_option_value (i, long_option, value)
         end if
         n_user_lib = n_user_lib + 1
         cycle SCAN_CMDLINE
      case ("--user-target")
         user_target = get_option_value (i, long_option, value)
         cycle SCAN_CMDLINE
      case ("--write-syntax-tables")
         call no_option_value (long_option, value)
         call init_syntax_tables ()
         call write_syntax_tables ()
         call final_syntax_tables ()
         stop
         cycle SCAN_CMDLINE
      case default
         call print_usage ()
         call msg_fatal ("Option '" // trim (arg) // "' not recognized")
   end select
end select
select case (arg(1:1))
case ("-")
   j = 1
   if (len_trim (arg) == 1) then
      look_for_options = .false.
   else
      SCAN_SHORT_OPTIONS: do
         j = j + 1
         if (j > len_trim (arg)) exit SCAN_SHORT_OPTIONS
         option = "-" // arg(j:j)
         select case (option)
         case ("-V")
            call print_version (); stop
         case ("-?", "-h")
            call print_usage (); stop
         case ("-e")
            command = get_option_value (i, var_str (option))
            call ifile_append (commands, command)
            cycle SCAN_CMDLINE
         case ("-i")
            interactive = .true.
            cycle SCAN_SHORT_OPTIONS
         case ("-l")
            if (j == len_trim (arg)) then
               library = get_option_value (i, var_str (option))
            else
               library = trim (arg(j+1:))
            end if
            libraries = libraries // " " // library
```

```
                        cycle SCAN_CMDLINE
                case ("-L")
                   if (j == len_trim (arg)) then
                      logfile = get_option_value (i, var_str (option))
                   else
                      logfile = trim (arg(j+1:))
                   end if
                   cycle SCAN_CMDLINE
                case ("-m")
                   if (j < len_trim (arg))  call msg_fatal &
                        ("Option '" // option // "' needs a value")
                   model = get_option_value (i, var_str (option))
                   cycle SCAN_CMDLINE
                case ("-r")
                   rebuild_library = .true.
                   rebuild_user = .true.
                   rebuild_phs = .true.
                   rebuild_grids = .true.
                   rebuild_events = .true.
                   cycle SCAN_SHORT_OPTIONS
                case ("-u")
                   user_code_enable = .true.
                   cycle SCAN_SHORT_OPTIONS
                case default
                   call print_usage ()
                   call msg_fatal &
                        ("Option '" // option // "' not recognized")
                end select
             end do SCAN_SHORT_OPTIONS
          end if
       case default
          files = files // " " // trim (arg)
       end select
    else
       files = files // " " // trim (arg)
    end if
end do SCAN_CMDLINE

! Overall initialization
if (logfile /= "")  call logfile_init (logfile)
if (banner)  call msg_banner ()

 ! Run any self-checks (and no commands)
 if (checks /= "") then
    checks = trim (adjustl (checks))
    RUN_CHECKS: do while (checks /= "")
       call split (checks, check, " ")
       call whizard_check (check, test_results)
    end do RUN_CHECKS
    call test_results%wrapup (6, success)
    if (.not. success)  quit_code = 7
    quit = .true.
 end if
```

```
! Set options and initialize the whizard object
allocate (options)
options%preload_model = model
options%default_lib = default_lib
options%preload_libraries = libraries
options%rebuild_library = rebuild_library
options%recompile_library = recompile_library
options%rebuild_user = rebuild_user
options%rebuild_phs = rebuild_phs
options%rebuild_grids = rebuild_grids
options%rebuild_events = rebuild_events

allocate (whizard_instance)
call whizard_instance%init (options, paths, logfile)

call mask_term_signals ()

! Run commands given on the command line
if (.not. quit .and. ifile_get_length (commands) > 0) then
   call whizard_instance%process_ifile (commands, quit, quit_code)
end if

if (.not. quit) then
   ! Process commands from standard input
   if (.not. interactive .and. files == "") then
      call whizard_instance%process_stdin (quit, quit_code)

      ! ... or process commands from file
   else
      files = trim (adjustl (files))
      SCAN_FILES: do while (files /= "")
         call split (files, this, " ")
         call whizard_instance%process_file (this, quit, quit_code)
         if (quit)  exit SCAN_FILES
      end do SCAN_FILES

   end if
end if

! Enter an interactive shell if requested
if (.not. quit .and. interactive) then
   call whizard_instance%shell (quit_code)
end if

! Overall finalization
call ifile_final (commands)

deallocate (options)

call whizard_instance%final ()
deallocate (whizard_instance)

call terminate_now_if_signal ()
call release_term_signals ()
```

```
      call msg_terminate (quit_code = quit_code)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
contains

  subroutine no_option_value (option, value)
    type(string_t), intent(in) :: option, value
    if (value /= "") then
       call msg_error (" Option '" // char (option) // "' should have no value")
    end if
  end subroutine no_option_value

  function get_option_value (i, option, value) result (string)
    type(string_t) :: string
    integer, intent(inout) :: i
    type(string_t), intent(in) :: option
    type(string_t), intent(in), optional :: value
    character(CMDLINE_ARG_LEN) :: arg
    character(CMDLINE_ARG_LEN) :: arg_value
    integer :: arg_len, arg_status
    logical :: has_value
    if (present (value)) then
       has_value = value /= ""
    else
       has_value = .false.
    end if
    if (has_value) then
       string = value
    else
       i = i + 1
       call get_command_argument (i, arg_value, arg_len, arg_status)
       select case (arg_status)
       case (0)
       case (-1)
          call msg_error (" Option value truncated: '" // arg // "'")
       case default
          call print_usage ()
          call msg_fatal (" Option '" // char (option) // "' needs a value")
       end select
       select case (arg(1:1))
       case ("-")
          call print_usage ()
          call msg_fatal (" Option '" // char (option) // "' needs a value")
       end select
       string = trim (arg_value)
    end if
  end function get_option_value

  subroutine print_version ()
    print "(A)", "WHIZARD " // WHIZARD_VERSION
    print "(A)", "Copyright (C) 1999-2014 Wolfgang Kilian, Thorsten Ohl, Juergen Reuter"
    print "(A)", "              --------------------------------------              "
    print "(A)", "This is free software; see the source for copying conditions.  There is NO"
    print "(A)", "warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE."
```

```
      print *
end subroutine print_version

subroutine print_usage ()
  print "(A)", "WHIZARD " // WHIZARD_VERSION
  print "(A)", "Usage: whizard [OPTIONS] [FILE]"
  print "(A)", "Run WHIZARD with the command list taken from FILE(s)"
  print "(A)", "Options for resetting default directories and tools" &
          // "(GNU naming conventions):"
  print "(A)", "     --prefix DIR"
  print "(A)", "     --exec_prefix DIR"
  print "(A)", "     --bindir DIR"
  print "(A)", "     --libdir DIR"
  print "(A)", "     --includedir DIR"
  print "(A)", "     --datarootdir DIR"
  print "(A)", "     --libtool LOCAL_LIBTOOL"
  print "(A)", "     --lhapdfdir DIR   (PDF sets directory)"
  print "(A)", "Other options:"
  print "(A)", "-h, --help          display this help and exit"
  print "(A)", "     --banner          display banner at startup (default)"
  print "(A)", "-e, --execute CMDS    execute SINDARIN CMDS before reading FILE(s)"
  print "(A)", "-i, --interactive     run interactively after reading FILE(s)"
  print "(A)", "-l, --library         preload process library NAME"
  print "(A)", "     --localprefix DIR"
  print "(A)", "                       search in DIR for local models (default: ~/.whizard)"
  print "(A)", "-L, --logfile FILE    write log to FILE (default: 'whizard.log'"
  print "(A)", "     --logging         switch on logging at startup (default)"
  print "(A)", "-m, --model NAME      preload model NAME (default: 'SM')"
  print "(A)", "     --no-banner       do not display banner at startup"
  print "(A)", "     --no-library      do not preload process library"
  print "(A)", "     --no-logfile      do not write a logfile"
  print "(A)", "     --no-logging      switch off logging at startup"
  print "(A)", "     --no-model        do not preload a model"
  print "(A)", "     --no-rebuild      do not force rebuilding"
  print "(A)", "-r, --rebuild         rebuild all (see below)"
  print "(A)", "     --rebuild-library"
  print "(A)", "                       rebuild process code library"
  print "(A)", "     --rebuild-user    rebuild user-provided code"
  print "(A)", "     --rebuild-phase-space"
  print "(A)", "                       rebuild phase-space configuration"
  print "(A)", "     --rebuild-grids   rebuild integration grids"
  print "(A)", "     --rebuild-events  rebuild event samples"
  print "(A)", "     --recompile       recompile process code"
  print "(A)", "-u --user             enable user-provided code"
  print "(A)", "     --user-src FILE   user-provided source file"
  print "(A)", "     --user-lib FILE   user-provided library file"
  print "(A)", "     --user-target BN  basename of created user library (default: user)"
  print "(A)", "-V, --version         output version information and exit"
  print "(A)", "     --write-syntax-tables"
  print "(A)", "                       write the internal syntax tables to files and exit"
  print "(A)", "-                     further options are taken as filenames"
  print *
  print "(A)", "With no FILE, read standard input."
end subroutine print_usage
```

```
    end program main
```

23.5   Whizard-C-Interface

⟨`whizard-c-interface.f90`⟩≡
  ⟨*File header*⟩

  ⟨*Whizard-C-Interface: Internals*⟩
  ⟨*Whizard-C-Interface: Init and Finalize*⟩
  ⟨*Whizard-C-Interface: Interfaced Commads*⟩
  ⟨*Whizard-C-Interface: HepMC*⟩

⟨*Whizard-C-Interface: Internals*⟩≡
```
    subroutine c_whizard_convert_string (c_string, f_string)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!

      implicit none

      character(kind=c_char), intent(in) :: c_string(*)
      type(string_t), intent(inout) :: f_string
      character(len=1) :: dummy_char
      integer :: dummy_i = 1

      f_string = ""
      do
         if (c_string(dummy_i) == c_null_char) then
            exit
         else if (c_string(dummy_i) == c_new_line) then
            dummy_char = CHAR(13)
            f_string = f_string // dummy_char
            dummy_char = CHAR(10)
         else
            dummy_char = c_string (dummy_i)
         end if
         f_string = f_string // dummy_char
         dummy_i = dummy_i + 1
      end do
      dummy_i = 1
    end subroutine c_whizard_convert_string

    subroutine c_whizard_commands (w_c_instance, cmds)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!
      use commands
      use diagnostics !NODEP!
      use lexers
      use models
      use parser
      use whizard
```

```fortran
      type(c_ptr), intent(inout) :: w_c_instance
      type(whizard_t), pointer :: whizard_instance
      type(string_t) :: cmds
      type(parse_tree_t) :: parse_tree
      type(parse_node_t), pointer :: pn_root
      type(stream_t), target :: stream
      type(lexer_t) :: lexer
      type(command_list_t), target :: cmd_list

      call c_f_pointer (w_c_instance, whizard_instance)
      call lexer_init_cmd_list (lexer)
      call syntax_cmd_list_init ()

      call stream_init (stream, cmds)
      call lexer_assign_stream (lexer, stream)
      call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
      pn_root => parse_tree_get_root_ptr (parse_tree)

      if (associated (pn_root)) then
         call cmd_list%compile (pn_root, whizard_instance%global)
      end if
      call cmd_list%execute (whizard_instance%global)
      call cmd_list%final ()

      call parse_tree_final (parse_tree)
      call stream_final (stream)
      call lexer_final (lexer)
      call syntax_cmd_list_final ()
    end subroutine c_whizard_commands
```

⟨*Whizard-C-Interface: Init and Finalize*⟩≡

```fortran
    subroutine c_whizard_init (w_c_instance) bind(C)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!
      use system_dependencies !NODEP!
      use limits, only: CMDLINE_ARG_LEN !NODEP!
      use diagnostics !NODEP!
      use unit_tests
      use ifiles
      use os_interface
      use whizard

      implicit none

      type(c_ptr), intent(out) :: w_c_instance
      character(CMDLINE_ARG_LEN) :: arg
      character(2) :: option
      type(string_t) :: long_option, value
      integer :: i, j, arg_len, arg_status
      logical :: look_for_options
      logical :: interactive
      logical :: banner
      type(string_t) :: files, this, model, default_lib, library, libraries
      type(string_t) :: check, checks, logfile
```

```fortran
type(test_results_t) :: test_results
logical :: success
logical :: user_code_enable = .false.
integer :: n_user_src = 0, n_user_lib = 0
type(string_t) :: user_src, user_lib
type(paths_t) :: paths
logical :: rebuild_library, rebuild_user
logical :: rebuild_phs, rebuild_grids, rebuild_events
logical :: recompile_library
type(ifile_t) :: commands
type(string_t) :: command

type(whizard_options_t), allocatable :: options
type(whizard_t), pointer :: whizard_instance

! Exit status
logical :: quit = .false.
integer :: quit_code = 0

! Initial values
look_for_options = .true.
interactive = .false.
files = ""
model = "SM"
default_lib = "default_lib"
library = ""
libraries = ""
banner = .true.
logging = .true.
logfile = "whizard.log"
check = ""
checks = ""
user_src = ""
user_lib = ""
rebuild_library = .false.
recompile_library = .false.
rebuild_user = .false.
rebuild_phs = .false.
rebuild_grids = .false.
rebuild_events = .false.
call paths_init (paths)

! Overall initialization
if (logfile /= "")  call logfile_init (logfile)
call mask_term_signals ()
if (banner)  call msg_banner ()

! Set options and initialize the whizard object
allocate (options)
options%preload_model = model
options%default_lib = default_lib
options%preload_libraries = libraries
options%rebuild_library = rebuild_library
options%rebuild_user = rebuild_user
```

```fortran
      options%rebuild_phs = rebuild_phs
      options%rebuild_grids = rebuild_grids
      options%rebuild_events = rebuild_events

      allocate (whizard_instance)
      call whizard_instance%init (options, paths)

      if (checks /= "") then
         checks = trim (adjustl (checks))
         RUN_CHECKS: do while (checks /= "")
            call split (checks, check, " ")
            call whizard_check (check, test_results)
         end do RUN_CHECKS
         call test_results%wrapup (6, success)
         if (.not. success)  quit_code = 7
         quit = .true.
      end if

      w_c_instance = c_loc (whizard_instance)

    end subroutine c_whizard_init

    subroutine c_whizard_finalize (w_c_instance) bind(C)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!
      use system_dependencies !NODEP!
      use limits, only: CMDLINE_ARG_LEN !NODEP!
      use diagnostics !NODEP!
      use ifiles
      use os_interface
      use whizard

      type(c_ptr), intent(in) :: w_c_instance
      type(whizard_t), pointer :: whizard_instance
      logical :: quit = .false.
      integer :: quit_code = 0

      call c_f_pointer (w_c_instance, whizard_instance)
      call whizard_instance%final ()
      deallocate (whizard_instance)
      call terminate_now_if_signal ()
      call release_term_signals ()
      call msg_terminate (quit_code = quit_code)
    end subroutine c_whizard_finalize

    subroutine c_whizard_process_string (w_c_instance, c_cmds_in) bind(C)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!

      implicit none

      type(c_ptr), intent(inout) :: w_c_instance
      character(kind=c_char) :: c_cmds_in(*)
      type(string_t) :: f_cmds
```

```
      call c_whizard_convert_string (c_cmds_in, f_cmds)
      call c_whizard_commands (w_c_instance, f_cmds)
    end subroutine c_whizard_process_string
```

⟨*Whizard-C-Interface: Interfaced Commads*⟩≡

```
    subroutine c_whizard_model (w_c_instance, c_model) bind(C)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!

      implicit none

      type(c_ptr), intent(inout) :: w_c_instance
      character(kind=c_char) :: c_model(*)
      type(string_t) :: model, mdl_str

      call c_whizard_convert_string (c_model, model)
      mdl_str = "model = " // model
      call c_whizard_commands (w_c_instance, mdl_str)
    end subroutine c_whizard_model

    subroutine c_whizard_library (w_c_instance, c_library) bind(C)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!

      implicit none

      type(c_ptr), intent(inout) :: w_c_instance
      character(kind=c_char) :: c_library(*)
      type(string_t) :: library, lib_str

      call c_whizard_convert_string(c_library, library)
      lib_str = "library = " // library
      call c_whizard_commands (w_c_instance, lib_str)
    end subroutine c_whizard_library

    subroutine c_whizard_process (w_c_instance, c_id, c_in, c_out) bind(C)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!

      implicit none

      type(c_ptr), intent(inout) :: w_c_instance
      character(kind=c_char) :: c_id(*), c_in(*), c_out(*)
      type(string_t) :: proc_str, id, in, out

      call c_whizard_convert_string (c_id, id)
      call c_whizard_convert_string (c_in, in)
      call c_whizard_convert_string (c_out, out)
      proc_str = "process " // id // " = " // in // " => " // out
      call c_whizard_commands (w_c_instance, proc_str)
    end subroutine c_whizard_process

    subroutine c_whizard_compile (w_c_instance) bind(C)
```

3216

```
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  type(c_ptr), intent(inout) :: w_c_instance
  type(string_t) :: cmp_str
  cmp_str = "compile"
  call c_whizard_commands (w_c_instance, cmp_str)
end subroutine c_whizard_compile

subroutine c_whizard_beams (w_c_instance, c_specs) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_specs(*)
  type(string_t) :: specs, beam_str

  call c_whizard_convert_string (c_specs, specs)
  beam_str = "beams = " // specs
  call c_whizard_commands (w_c_instance, beam_str)
end subroutine c_whizard_beams

subroutine c_whizard_integrate (w_c_instance, c_process) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_process(*)
  type(string_t) :: process, int_str

  call c_whizard_convert_string (c_process, process)
  int_str = "integrate (" // process //")"
  call c_whizard_commands (w_c_instance, int_str)
end subroutine c_whizard_integrate

subroutine c_whizard_matrix_element_test &
     (w_c_instance, c_process, n_calls) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  integer(kind=c_int) :: n_calls
  character(kind=c_char) :: c_process(*)
  type(string_t) :: process, me_str
  character(len=8) :: buffer

  call c_whizard_convert_string (c_process, process)
  write (buffer, "(I0)")  n_calls
```

```fortran
    me_str = "integrate (" // process // ") { ?phs_only = true" // &
         "  n_calls_test = " // trim (buffer)
    call c_whizard_commands (w_c_instance, me_str)
  end subroutine c_whizard_matrix_element_test

  subroutine c_whizard_simulate (w_c_instance, c_id) bind(C)
    use, intrinsic :: iso_c_binding
    use iso_varying_string, string_t => varying_string !NODEP!

    implicit none

    type(c_ptr), intent(inout) :: w_c_instance
    character(kind=c_char) :: c_id(*)
    type(string_t) :: sim_str, id

    call c_whizard_convert_string(c_id, id)
    sim_str = "simulate (" // id // ")"
    call c_whizard_commands (w_c_instance, sim_str)
  end subroutine c_whizard_simulate

  subroutine c_whizard_sqrts (w_c_instance, c_value, c_unit) bind(C)
    use, intrinsic :: iso_c_binding
    use iso_varying_string, string_t => varying_string !NODEP!

    implicit none

    type(c_ptr), intent(inout) :: w_c_instance
    character(kind=c_char) :: c_unit(*)
    integer(kind=c_int) :: c_value
    integer :: f_value
    character(len=8) :: f_val
    type(string_t) :: val, unit, sqrts_str

    f_value = c_value
    write (f_val,'(i8)') f_value
    val = f_val
    call c_whizard_convert_string (c_unit, unit)
    sqrts_str = "sqrts =" // val // unit
    call c_whizard_commands (w_c_instance, sqrts_str)
  end subroutine c_whizard_sqrts
```

⟨*Whizard-C-Interface: HepMC*⟩≡

```fortran
    type(c_ptr) function c_whizard_hepmc_test &
         (w_c_instance, c_id, c_proc_id, c_event_id) bind(C)
    use, intrinsic :: iso_c_binding
    use iso_varying_string, string_t => varying_string !NODEP!
    use commands
    use diagnostics !NODEP!
    use events
    use hepmc_interface
    use lexers
    use models
    use parser
    use processes
```

```
use rt_data
use simulations
use whizard
use os_interface

implicit none

type(c_ptr), intent(inout) :: w_c_instance
type(string_t) :: sim_str
type(parse_tree_t) :: parse_tree
type(parse_node_t), pointer :: pn_root
type(stream_t), target :: stream
type(lexer_t) :: lexer
type(command_list_t), pointer :: cmd_list
type(whizard_t), pointer :: whizard_instance

integer :: i_evt
type(simulation_t), target :: sim

character(kind=c_char), intent(in) :: c_id(*)
type(string_t) :: id
integer(kind=c_int), value :: c_proc_id, c_event_id
integer :: proc_id, event_id

type(hepmc_event_t), pointer :: hepmc_event

type(process_t), pointer :: process
integer :: proc

integer :: factorization_mode, try

call c_f_pointer (w_c_instance, whizard_instance)

call c_whizard_convert_string (c_id, id)
sim_str = "simulate (" // id // ")"

proc_id = c_proc_id
event_id = c_event_id

allocate (hepmc_event)
call hepmc_event_init (hepmc_event, c_proc_id, c_event_id)

call syntax_cmd_list_init ()
call lexer_init_cmd_list (lexer)

call stream_init (stream, sim_str)
call lexer_assign_stream (lexer, stream)
call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
pn_root => parse_tree_get_root_ptr (parse_tree)

allocate (cmd_list)
if (associated (pn_root)) then
   call cmd_list%compile (pn_root, whizard_instance%global)
end if
```

```
      call sim%init ([id], .true., .true., whizard_instance%global)

      !!! This should generate a HepMC event as hepmc_event_t type
      call msg_message ("Not enabled for the moment.")

      call sim%final ()

      call cmd_list%final ()

      call parse_tree_final (parse_tree)
      call stream_final (stream)
      call lexer_final (lexer)
      call syntax_cmd_list_final ()

      c_whizard_hepmc_test = c_loc(hepmc_event)
      return
    end function c_whizard_hepmc_test
```

Chapter 24

Cross References